Improving usability in pan gateways by means of a novel Bluetooth pairing method

Regan, Philippa

IMPROVING USABILITY IN PAN GATEWAYS BY MEANS OF A NOVEL BLUETOOTH PAIRING METHOD

Submitted for the degree of Master of Science, Durham University, 21st November 2002.

Philippa Regan

Abstract

This thesis investigates the usability issues surrounding an implementation of the Personal Area Network (PAN) Gateway, a new concept in mobile communications. The PAN Gateway device consists of a GSM/GPRS modem and a Bluetooth modem. The Bluetooth modem is used to link mobile devices to form a PAN and the GSM/GPRS modem is used to link the PAN to external networks.

The possible Man Machine Interfaces for the PAN Gateway are discussed together with the usability of existing Bluetooth devices. A weakness was discovered in the usability and security of Bluetooth Pairing in existing mobile devices and this led to the development of the "Touch and Find" system and the Pairing Link Protocol. The "Touch and Find" system interacts with the Bluetooth stack and allows simple, intuitive pairing of Bluetooth devices via a serial link. A full duplex serial link was implemented using simple electrical contacts to provide the link. Inductive coupling and infrared solutions were also developed. The Pairing Link Protocol specifies the signal flow for the "Touch and Find" process. The "Touch and Find" system that was implemented using simple electrical contacts shows how simple Bluetooth pairing can be. Pairing is simply carried out by briefly touching together the devices to be paired.

The "Touch and Find" system was implemented in C on Borland C++ and used in conjunction with TTPCom's Bluetooth development system, which consists of a "Mad Cow" evaluation board and Genie – a Bluetooth development tool.

The research carried out demonstrates the feasibility of the "Touch and Find" system over a variety of physical mediums. The system greatly improves the usability of Bluetooth Pairing, thus improving the "Out of Box" experience. It is likely that the Inductive solution can be extended to enable battery charging across the "Touch and Find" Inductive interface, further enhancing the "value added" capabilities of this system.

IMPROVING USABILITY IN PAN GATEWAYS BY MEANS OF A NOVEL BLUETOOTH PAIRING METHOD

Philippa Kate Regan

This thesis is submitted in candidature for the degree of Master of Science in the School of Engineering, University of Durham, 21st November 2002.

Declaration

The material contained within this thesis is the sole work of the author and has not been submitted previously in this or any other University.

Statement of Copyright

ACKNOWLEDGEMENTS

The author wishes to thank TTPCommunications, Cambridge for their support throughout this project.

The author wishes to thank the following people for their help and patience: -

From the School of Engineering, Durham University: -
Peter Baxendale, Ian Hutchison, Dr. Simon Johnson, Richard McWilliam, William Pugsley and Craig Robinson.

From TTPCommunications, Cambridge: -
Gordon Aspin, Andy Fogg, Jerome Guibal, Dr. John Haine, Dr. David Kyle, Matthew Waterson, Neil Werdmuller.

Additionally, the author wishes to thank Joanna Donkin, Keith Herrman, Jennifer Regan and Michael Regan for their continuing support.

TABLE OF CONTENTS

TABLE OF FIGURES

GLOSSARY

AFH	Adaptive Frequency Hopping
ATOMIK	Alphabetically Tuned & Optimised Character Layout
BWIG	Bluetooth Wireless Internet Gateway
DSSS	Direct Sequence Spread Spectrum
FCC	Federal Communications Commission
FHSS	Frequency Hopping Spread Spectrum
FOCC	Fluctuating Optimal Character Layout
GPRS	General Packet Radio Service
Graffiti	Input method for palm pilots using a stylus
GSM	Global System for Mobile Communication
HCI	Human Computer Interaction
IrDA	Infrared Data Association
ISM	Industrial Scientific Medical
LoS	Line of Sight
Mbps	Mega bits per second
MMS	Multi-media Messaging Service
PAN	Personal Area Network
PDA	Personal Digital Assistant
RF	Radio Frequency
SHORE	Student HCI Online Research Experiments
SMS	Short Message Service
T9	Tegic 9 – predictive text system.
TCP/IP	Transfer Control Protocol/Internet Protocol
Wpm	Words per Minute

CHAPTER 1 INTRODUCTION

The research investigated how to improve the usability of the Bluetooth Pairing process used within mobile devices in order to improve the user's "Out of Box" experience. Initially the research focused on the concept of a Personal Area Network (PAN) Gateway device and how the user interface should be designed, this led to an examination of various methods of text entry and an investigation into the usability of existing Bluetooth enabled mobile 'phones. The project was undertaken in conjunction with TTPCommunications (TTPCom), Cambridge.

A clear shortfall in the usability and security of Bluetooth pairing was discovered and it was decided to develop the "Touch and Find" system to overcome these problems. The "Touch and Find" system uses a serial link to transfer the data required to enable the pairing of Bluetooth devices. The "Pairing Link Protocol" was designed to specify the signal flow required in order to exchange the necessary information.

The software for the "Touch and Find" system was developed in C, using Borland C++ and TTPCom's "Genie" emulator in addition to a TTPCom "Mad Cow" Bluetooth Evaluation Board. The "Touch and Find" system interfaces with the Device Manager within the TTPCom implementation of the Bluetooth stack and transmits the signals specified in the "Pairing Link Protocol" from a Windows serial port to the developed hardware circuitry for transmission.

The investigation into which physical mediums the "Touch and Find" system could use has resulted in the successful implementation and testing of a simple electrical contact solution and an inductive solution in addition to the design of two Infrared solutions.

The "Touch and Find" system developed clearly shows the feasibility of using this method to increase the usability and security of Bluetooth pairing using a variety of physical mediums. The "Pairing Link Protocol" has enabled a robust link to be formed to exchange the necessary information.

1.1 INTRODUCTION TO THE THESIS

Chapters 2 and 3 contain the literature survey for the thesis. Chapter 2 introduces the PAN Gateway concept and how it might be used. It goes on to discuss the usability of mobile devices and in particular the various existing text entry methods that have been designed. The chapter finishes with an evaluation of the text entry methods covered and a summary of what has been covered in the chapter.

In Chapter 3 the wireless technologies which could be used to provide the local area connectivity for the PAN Gateway are introduced and evaluated together with relevant coexistence issues. The usability of Bluetooth in existing mobile devices is evaluated and the future of Bluetooth considered.

In Chapter 4 the aims of the PAN Gateway are discussed and possible user interfaces are considered, evaluated and an optimal interface selected. The general requirements of the PAN Gateway are stated and a new concept to improve the usability of Bluetooth Pairing is introduced.

Chapter 5 develops the concept proposed in Chapter 4 to improve the usability in Bluetooth pairing into the "Touch and Find" system. The chapter begins with an overview of the aims, concepts and requirements of the "Touch and Find" system and then moves on to describe and evaluate the two different concepts developed by the author to specify the signal flow (the Pairing Link Protocol) required in the "Touch and Find" system. The design of the main PLP task code is described and any important decisions made in the design process highlighted. Finally the implementation and testing of the main PLP task is documented.

In Chapter 6 the PLP transport task is described. The chapter begins with describing where the PLP transport task (PLPTX) fits into the "Touch and Find" system and its requirements. A thorough description of how to set up and use the Windows serial port for non-overlapped Input/Output is given. The development of the methods to write code to the serial port and read code from it is described together with a discussion about the signal structure and how its design affects the processing task in the receive sections. Finally the successful implementation and testing of the PLP transport task is described.

In Chapter 7 the hardware is developed, putting into place the final part of the "Touch and Find" system. The chapter starts with an introduction to the simple electrical

contact solution which uses the "hybrid" transceiver and describes the modifications required to the software in order to detect the connected state. The designs for two infrared solutions are described and evaluated. Finally, the design of the "four coil" Inductive coupling solution is described.

Chapter 8 contains the evaluation, conclusion and summary for the work and the thesis. The software, hardware and the complete system are evaluated. The content of the thesis and conclusions drawn from the chapters are summarised and then overall conclusions given. Finally possible enhancements and further work for the "Touch and Find" system are described and then a concluding statement is given.

CHAPTER 2 THE PAN GATEWAY DEVICE

This chapter introduces the concept of the PAN Gateway and shows the context in which the PAN Gateway might be used. The usability of mobile devices is discussed together with a comprehensive description of existing text entry methods.

2.1 THE PAN GATEWAY CONCEPT

A Personal Area Network (PAN) is a collection of devices that might be carried by a mobile, networked individual (for example a professional on the move, an internet-wise tourist)[1]. The devices may include any subset of: cell phone, laptop, mobile 'phones, palm pilot and other mobile devices.

The envisaged PAN (Personal Area Network) Gateway would be a small device that would contain a GSM/GPRS modem and a Bluetooth modem, as shown in Figure 2-1. The GSM/GPRS modem would be used to connect the PAN to external networks. The Bluetooth modem is used to enable connections between local Bluetooth enabled devices such as PDA's (Personal Digital Assistant), cameras and PC's. "Bluetooth is a low cost, low power, short-range radio technology, originally developed as a cable replacement to connect devices such as mobile 'phone handsets, headsets, and portable computers" [2]. Other technologies that could have been used to provide local connectivity will be discussed in Chapter 3.

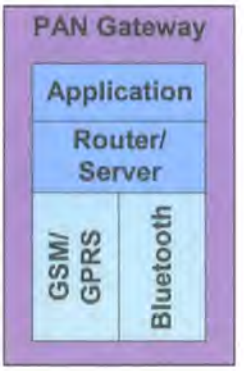


Figure 2-1 Block Diagram of PAN Gateway.

Rather than simply using a Bluetooth enabled mobile 'phone it is advantageous to simplify the interfaces on the PAN gateway in order to create a small, simple, cost effective, user friendly device. Ideally the PAN gateway will be the size of a matchbox and would only require an On/Off button and an L.E.D. to indicate network coverage and power on.

Bluetooth (see section 3.2.5 for more information) is not a line-of-sight connection and so the PAN gateway device could be left in a briefcase or attached to the users belt for example. There are many factors that need to be considered in the design of a PAN Gateway, such as whether the user will always have another device that they can use to dial the required number for a 'phone function and how to enable Bluetooth pairing. These factors are considered in Chapter 4. The usability of existing devices is discussed in section 2.3 (Usability in Mobile Devices) and the technological aspects are discussed in Chapter 3.

The PAN gateway system has significant advantages; the use of Bluetooth means that full functionality can be obtained from your PDA wherever you are in the world. It can link up to any Bluetooth enabled handset and is not restricted to countries with compatible mobile 'phone technologies. Upgrading your PDA will be cheaper as you'll be replacing a simpler device, as it does not have the GSM functions in it. In a white paper for Lucent Technologies, Wetzel stated [3] that Bluetooth wireless technology in PAN's is very likely to change the way we handle and access data in the near future. He noted that a similar development during the past ten years can be observed in the way the mobile phone has changed our behaviour in terms of information vs. independence of location.

In addition, the PAN Gateway concept is advantageous for network operators. Network operators presently massively subsidise the cost of handsets based on the revenue that will be gained from use of the handset, i.e. "line rental" and call charges. As mobile 'phones become more integrated, operators are being expected to provide users with increasingly expensive handsets for minimal cost. Meanwhile, the mobile 'phone market has become significantly more competitive, reducing the revenue from each call made. The PAN gateway could allow a move away from this increasingly undesirable situation by offering a new concept in mobile communication. Initially it will probably be necessary for operators to include a basic Bluetooth enabled handset to enable "traditional" voice calls.

A similar concept to the PAN Gateway is BWIG (Bluetooth Wireless Internet Gateway) [4] which defines a Bluetooth usage model that provides seamless ad-hoc web access for internet enabled mobile devices through a Bluetooth enabled fixed internet connection; Users will be able to access the internet without the need for a dialup connection, providing faster throughput speeds (11Mbps can be achieved on aggregate using a fully deployed Scatternet) wirelessly and with the capability to share

the connection. In BWIG the TCP/IP protocols are not used over the Bluetooth Radio in order to reduce the overhead across the Bluetooth link.

Another similar concept is being developed by Norwood Systems [5], their aim is to provide voice applications throughout the office to create a local network for voice communications using low power devices such as mobile 'phones, cordless headsets and voice enabled PDA's. The system will incorporate a voice recognition server and telephony gateway allowing on screen dialling, voice recognition for calling from a headset, dictation via headset and other functions.

2.2 The Need for a PAN Gateway

According to a European Commission study carried out in Spring 2000, 55% of the population own a Mobile 'phone in Europe. Two years later, ownership of a mobile 'phone has increased and in many European countries there are now more mobile 'phone subscriptions than fixed line subscriptions. As usage of mobile 'phones for voice applications increases so does the usage for data applications.

More and more 'phones now come with built in support for data functions; increasingly consumers are able to synchronise their mobile 'phone with their PC using a serial, USB, IrDa or even Bluetooth connection. As the functions of mobile 'phones, PDA's and PC's continue to merge, the question arises as to whether mobile 'phones with integrated address, diary and data functions or PDA's with mobile communications capabilities will become the dominant market force. Despite the continuing convergence of the mobile 'phone and the PDA it is clear that at present the two devices have quite different features; display size, input methods, software, services available, and communication with other services distinguish the devices currently on offer [6].

Presently the majority of "organiser" functions that are included on mobile 'phones are not very user friendly or indeed particularly useful, although there are increasingly more exceptions. It seems that PDA manufacturers Palm and Handspring are beating many of the mobile 'phone manufacturers at their own game by creating integrated 'phone's/PDA's. For example, the Handspring Visor 'phone has a large screen and provides multiple ways of accessing the 'phone book function.

Usability expert Jacob Nielson [7] stated that there is a major problem in the different form factors required for each of the two sets of functions. i.e. A device that is suited to

holding up to your ear is not a device well suited to accessing the Internet (at least until the Internet becomes voice powered), whereas a PDA has a large screen and is thus well suited to accessing and viewing data. Similarly, holding a PDA to your ear does not feel good, although perhaps users will become more accustomed to doing just that or alternatively using a headset in the same way that users overcame the initial embarrassment of talking on a mobile 'phone in public.

For business users there is clearly a need for a GSM modem to be attached to a PDA – it allows web surfing in a more user friendly way than using a mobile 'phone, allows emails to be sent and received and other similar tasks. Up until now the products that have been developed focus on integrating the GSM modem into the PDA or vice versa. For example Nokia have created the Communicator series, more recently the 9210, Palm have created the VII series of Palm computers that have an integral GSM modem to allow such functions and have produced "clip on" GSM modems for their other devices. These products require a monthly subscription which means that the user is now paying for both a mobile 'phone subscription and a Palm subscription as well as carrying around two GSM modems!

Handspring have taken it a step further in their Treo series which "combines a 'phone, a pager and an organizer into one small product so people could carry a single device instead of two or three" said Jeff Hawkins, founder of Handspring [8].

There are several problems with such integrated devices, for example: -

- If you go out for a walk and want to take your 'phone with you for safety reasons, do you really want to risk losing your PDA with all your personal information on it.
- To be usable for web surfing the screen needs to be big, making the device bulky to carry around. If the device is easy to carry, the interface will have to be smaller and less usable.
- When new technology is released, are the users going to be prepared to pay several hundred dollars for a new system – when all they need is a small modification – for example GPRS capability.

For a business user the only alternative to an integrated device is to carry around a mobile 'phone, a PDA and a PC (for writing longer messages). There is clearly a need for a different type of device, such as a device that could act as a central hub for communications between all of an individuals communications devices and the outside world. Although it could be argued that this already exists in the form of a mobile 'phone, a more revolutionary concept should be considered.

2.3 Usability in Mobile Devices

The main usability aspect discussed here is methods of text entry, as text entry is one of the most limiting factors in the usability of a mobile device. However, there are many other factors that influence how usable a mobile 'phone is, including the form factor size, shape and weight, the user interface (see Chapter 4) the screen size, the battery and the network services. The usability of a device is fundamental to the popularity of the device although clearly marketing plays a significant role as well.

It has been suggested [9] that one crucial factor in Nokia being the world's leading maker of mobile 'phones is their "user-centred approach to developing products". In a usability evaluation between the Nokia 3210 and the Siemens C25, the following factors were found to influence the usability of the devices.

- Complexity – the Nokia uses only three buttons to do what the Siemens requires five buttons.
- Consistency – Button functions on the Nokia are consistent with dedicated buttons for selecting, scrolling and cancelling - making the device easier to use.
- Clutter – the Siemens 'phone makes heavy use of icons – however these are not readily interpretable or noticeable because of their small size and varying locations on the display area.

Overall it was concluded that the Siemens 'phone makes more options available at any one time, but at considerable cost in complexity, consistency and clarity making the Siemens 'phone significantly more complex to learn. Another advantage of Nokia 'phones is their consistency, when users upgrade they know that the user interface on the new 'phone will be similar to that on their old 'phone, making them confident that they will be able to use it easily.

Matthias suggests that energy storage is the major technological hurdle to be solved in mobile devices [10]. This is evidenced by the increasing drive to design very low power systems for use in mobile devices in order to preserve battery life. After all, a mobile phone is not very "mobile" if you have to charge it every few hours. Low power consumption was a fundamental factor in the design of the Bluetooth specification and is one of the main reasons for Bluetooth's dominance over IEEE802.11b in mobile devices. Advances have been made in battery technology with the creation of the Lithium Polymer battery. These batteries have a high energy density, so smaller batteries can deliver more power for longer. Importantly these batteries are also

shapeable and so can be designed to fit inside an ergonomic form of the device. These batteries are used in the Palm m500 and m505 PDA's.

2.4 METHODS OF TEXT ENTRY

A major limitation on the usability and usage of mobile 'phones is the available text entry methods. As yet, manufacturers of handheld devices have not successfully duplicated the functionality of the desktop keyboard; present methods of text entry give similar input speeds of 10 –15 wpm[1] to those available hundreds of years ago with pen and ink compared to 50+ wpm with a desktop keyboard [11].

A recent study clearly demonstrated the value of handheld computers in the home, but the lack of an efficient method of text entry made even basic tasks such as email and web surfing very difficult [12].

Although users have become accustomed to sending SMS[2] (text messages) using the triple tap method or alternatively T9 (Tegic 9) predictive text, most users would find the existing numerical interface too tedious and inefficient to use for messages that are much longer than the 160 characters in an SMS. Even the Graffiti™ system and the touch screen QWERTY keyboard used by Palm are too slow for long messages in addition to the user needing to learn a new alphabet (for using the Graffiti handwriting recognition technology).

In order for mobile email and Internet services to become truly user friendly, either a new approach must be taken to text entry or alternatively new ways to link existing interfaces must be considered. A particularly challenging problem for human factors researchers has been to develop alternative text entry methods using on screen keyboards for use by the "walk up" market; consumers who want to be able to use it immediately with little or no training.

There are two available methods for shrinking the size of a physical keyboard, the first is to shrink the size of each key, as done in electronic dictionary's; unfortunately this significantly slows the input speed due to the difficulty in using small keys. The second is to reduce the number of keys by giving each key a multiple use as in the mobile

[1] Wpm – Words per Minute

[2] SMS – Short Message Service

telephone keypad. However, this approach results in ambiguity which reduces the usability of such a device [13].

An investigation of text entry methods for mobile devices carried out by the author has identified many different methods. The methods outlined below range from QWERTY and non-QWERTY keyboards to dynamic character layout keyboards in addition to handwriting recognition (such as Graffiti™), voice recognition, triple tap, predictive text and the Alphagrip™.

2.4.1 Danger's "Hiptop"

The Danger "Hiptop" (Figure 2-2) is both a PDA and a cellular 'phone, eliminating the need to carry two devices. Usability is maximised by making the entire surface area devoted to screen area with the exception of a few thumb-operated buttons. The device has two different form factor configurations; the first is the size of a fat deck of cards in which you can only see the screen – this base form factor works fine for checking appointments or incoming mail. The second configuration is used to respond to email; the device twists open like a Russian snuffbox and reveals an QWERTY keyboard under the screen [14] [15].

Figure 2-2 Danger's Hiptop Device

One of the advantages of the "Hiptop" is that the display area is not reduced by the use of a soft keyboard. In addition, the physical size and shape of the "Hiptop" enables two-handed operation on the move by using the users thumbs to type. The two

handed approach makes good use of the QWERTY keyboard layout that was designed for two-handed operation.

2.4.2 Graffiti vs. on screen keyboard

Graffiti™ is a handwriting recognition technology used in Palm PDA's. Users of Graffiti™ are required to learn a slightly different alphabet in order for recognition of text to be achieved. An area of the touch screen on the Palm pilot is reserved for use of Graffiti™. Users write one letter at a time (using the modified alphabet) in the Graffiti™ area with a stylus designed for use on touch screens.

The SHORE [16] 2001 study used subjects with no previous experience of either keyboard tapping or Graffiti and measured the number of errors that the subjects made while carrying out a number of specified tasks. Unsurprisingly during the first trial block, using the keyboard led to faster entry and fewer errors than using Graffiti. In later trial blocks the time taken for users to complete tasks using Graffiti decreased to the point where it was almost the same as using the on screen keyboard. However the number of errors made using Graffiti remained significantly higher. LaLomia [17] observed that users are only willing to accept error rates of approximately 3%, a significant problem for Graffiti™.

The SHORE study showed that keyboard entry is faster and less error prone. The main problem with keyboard tapping is the excessive number of screen switches required during a common application. The researchers also stated that they believed that experienced Palm users would find usage of Graffiti significantly faster due to the delays encountered when making the necessary screen switches. Another study [13] has found that an experienced user of Graffiti™ on a Palm pilot can write about 20 words per minute, far slower than a well practiced typist on a physical keyboard. Text entry speeds for QWERTY keyboards implemented on a touch screen as used by Palm and other PDA manufacturers were determined to be around 30wpm [13].

Both the Graffiti™ handwriting recognition and the touch screen keyboards result in text entry speeds considerably lower than the 50 wpm that is typically produced on a physical keyboard, showing that further development is required in text entry methods although both match the speed of handwriting which is approximately 15-20wpm.

2.4.3 Soft Keyboards

QWERTY soft keyboards give a text entry speed of around 30wpm [13]. The primary advantage of a QWERTY layout is that users are familiar with the layout and therefore do not need to learn the new layout. However, the QWERTY keyboard is particularly inefficient for stylus tapping as it was designed for use with two hands as in conventional typing, thus adjacent letter pairs (digraphs) are placed on opposite sides of the keyboard. In addition, the QWERTY layout was designed when traditional typewriters were used; the QWERTY layout was designed to reduce the number of times the keys hit each other and got jammed. Another disadvantage of the QWERTY layout is that it is not international. This causes confusion for those who use PC's in different countries - it is even different between the UK and the U.S!

For stylus operated keyboards, digraphs should be placed next to each other as only one stylus is used [13]. Performance Modelling is used in the design of Virtual keyboards. The keyboard is optimised so that the typical total distance travelled to reach a key is minimised [13], i.e. that the most frequently used keys should be placed in the centre of the keyboard. Performance modelling is used to optimise the layout for a given language.

The QWERTY keyboard was analysed using a "sub-optimal" method and the performance found to be 30.5 wpm assuming that the user always taps on the portion of the space bar that minimises the character-space-character path [13]. Various non-QWERTY keyboards have been designed using performance modelling, some examples have been outlined below: -

Opti - (shown in Figure 2-3) was designed by MacKenzie and Zhang [18] and later modified by the designers to create OPTI2. Analysis of the OPTI2 keyboard showed that the performance was around 36 to 40 wpm, a considerable improvement over the QWERTY keyboard [13].

Figure 2-3 OPTI high performance soft keypad

Atomik - (Alphabetically Tuned and Optimised Mobile Interface Keyboard) is a highly optimised method using touch keyboards for entering data into handheld devices. ATOMIK has the potential to allow text entry speeds of over 40 words per minute using a touch screen keyboard [13].

The Atomik keyboard has been optimised by increasing the movement efficiency, alphabetically tuning the layout (letters are generally alphabetically ordered A to Z from top left to bottom right) to reduce the learning time required by novice users and by connecting commonly used words or fragments of words, see Figure 2-4 Atomik Keyboard Layout.

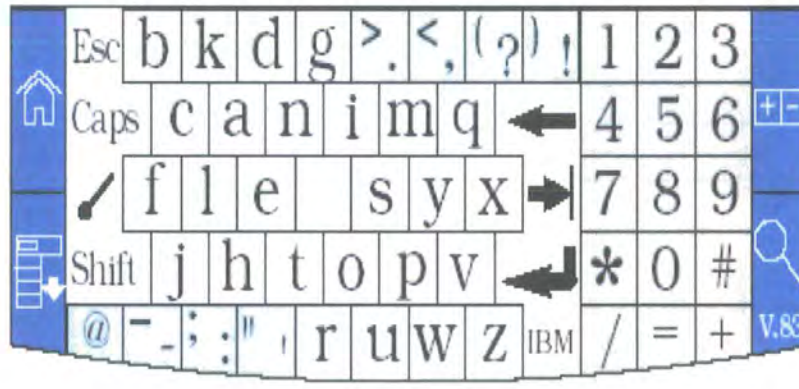


Figure 2-4 Atomik Keyboard Layout

This optimisation was achieved using a "Metropolis Optimization Algorithm" in which each key is treated as an atom and the keyboard is treated as a molecule. The "atomic" interactions between each of the keys drove the movement efficiency, defined

by the summation of all Fitts's[3] law movement times between every pair of keys, weighted by the statistical frequency of the pair of letters in English.

Other keyboards designed and analysed [13] are: -

- **FITALY** (Textware Solutions) - performed at about 36 wpm.
- **Chubbon** - performed at around 32 wpm.
- **Metropolis** – the Metropolis keyboard was designed using the Metropolis random walk method and performed at 43 wpm, the fastest of the above keyboards.

FOCL (Fluctuating Optimal Character Layout) is a concept developed by Bellmann and MacKenzie for use in "small, input-limited devices in mobile situations" in which the existing input method is arrow keys moving around a cursor on the character set displayed on a 3 or 4 line liquid crystal display. After each character "c" is entered the layout is rearranged so that the most likely next characters are closer to the cursor. Each new layout is optimal with respect to "c", but as the layout is perpetually changing there is a time penalty due to the user having to visually search for the correct character. FOCL does significantly reduce the mean number of keystrokes per character, however when tested against the QWERTY keyboard there were no significant differences in data entry speeds which were approximately 10 –15 wpm [19].

Soft keyboards clearly provide a good method for text entry in mobile devices - the best keyboard layouts to date are the Opti, Atomik and Metropolis which each perform at around 40 wpm. It is interesting to note that the FOCL concept, which is intuitively a bad idea, performs at a similar level to the widely used QWERTY keyboard at around 15wpm.

2.4.4 Alpha Grip

The Alphagrip™ was designed to create a new, faster input interface to replace slow, tedious text entry technologies and to ensure that productivity is not sacrificed in favour of portability. The Alphagrip™ is designed to provide a single flexible interface. The Alphagrip™ interfaces to other devices by means of both wired and wireless connections.

[3] Fitts' Law is a model used to account for the time it takes to point at something, based on the size and distance of the target object. Fitts' Law and variations of it are used to model the time it takes to use a mouse and other input devices to click on objects on a screen.

Alphagrip is a two-handed touch typing technology which is device agnostic and features a vertical hand orientation allowing the fingers to fall naturally and comfortably on full-sized, multi-directional buttons located on the back of the device. By requiring minimal movement users can quickly teach themselves to generate all the letters of the alphabet as well as punctuations. A mode switch button allows users to enjoy the functionality of several (otherwise dedicated) devices (keyboard/mouse, smart 'phone, PDA, game controller, or TV remote) within one form factor [13].

Alphagrip™ claims to "allow users to enter text quickly (50+wpm), easily and comfortably on full-sized keys regardless of the user's body position or the availability of a flat work surface" [20].

However, the Alphagrip™ will require a substantial amount of initial learning, providing an initial obstacle that may prevent users from learning to use the new technology. In recognition of this the manufacturers are initially targeting young people – the first prototypes incorporate the Alphagrip™ interface to a game controller. The manufacturers hope that learning to type on an Alphagrip™ will be "a game" to young people and that this will allow them to reach more users.

Figure 2-5 Alphagrip

Another major problem with the Alphagrip™ is its size as shown in Figure 2-5. The sheer size of the Alphagrip™ means that it is not conducive to portability. However its use as a single interface may be more beneficial. Once again the key lies in persuading users to invest the time in learning to use the device.

2.4.5 Modo

Modo (as shown in Figure 2-6) is a small device designed for accessing information specific to your location such as entertainment information and "going out" listings. Modo is operated by a single hand; the index finger rests on the back button and the users thumb rests on a wheel. The wheel has two functions; scrolling the wheel moves the selection up and down the screen and pressing the wheel activates the current selection [21]. The major advantage of the Modo is that it only requires one-handed operation, enabling you to use your other hand for carrying your briefcase or other items. The egg shaped design is good aesthetically and the device fits into the palm of your hand comfortably.

Figure 2-6 Modo device.

2.4.6 Triple tap, Two-key method or T9?

There are three methods of text entry based on the traditional 12 key keypad (shown in Figure 2-7); triple tap, two key input and T9 predictive text. Each of these methods is described below: -

Figure 2-7 'Phone Keypad

Triple Tap

To use the triple tap method, each key must be pressed one or more times by the user to specify a character, for example to type an "A" the number "2" key must be pressed

once, whereas to type a "C" the "2" key must be pressed three times. This approach brings out a problem with segmentation; when a character is on the same key as the previously entered character (as in the word "on") the system must determine if the new key press "belongs to" a new character or the previous character [22]. This is solved either by means of a timeout period or alternatively by pressing a key that forces the system to move onto the next key. The predicted expert rate performance of the triple tap method is approximately 21 – 27 wpm [22].

Two-key input method

In the two key input method the user selects the group of letters (for example key "2" gives "abc") and then a second number key is pressed for disambiguation, in this case "3" could be pressed to give the letter "c". The two key method is not very common but is very simple and requires no timeouts [22]. The predicted expert rate performance of the two-key input method is approximately 21 – 27 wpm [22].

T9

T9 (patented by Tegic Communication, Inc) text recognition uses a built in dictionary and adds knowledge to the system itself. It requires only one keystroke per letter and uses a built in dictionary for disambiguation. The "0" key is used for "SPACE". The "0" key is also used to delimit the word and terminate the disambiguation of proceeding keys. However this creates problems as multiple words may have the same key sequence, T9 then guesses the most common word and then allows users to press a key to view the next possible word [23]. The disambiguation is incorrect in around 5% of cases [22]. The predicted expert rate performance of the T9 system is 41 wpm for one handed thumb entry to 46 wpm for two handed index finger operation [22].

Text entry methods are compared using accuracy and speed of text entry. However, it should be noted that performances of expert and novice users are very different. The novice using T9 for the first time will have to go through some initial training and practice whereas the expert user will already be adept.

The SHORE 2001 (Student HCI[4] Online Research Experiments) study showed that usage of the T9 system had a longer learning time leading to frustration but that user satisfaction ratings were significantly higher once the initial training had been completed as it saved time. It is interesting to note that despite the initial frustration that users felt when using T9 that they were found to be more likely to buy a 'phone

[4] HCI – Human Computer Interaction

which had T9. This shows that users are prepared to cope with some frustration initially if it leads to a faster performance time in the end.

2.4.7 *Digit Wireless's Fastap™*

Fastap™ is a full keyboard deployed on a telephone handset as shown in Figure 2-8 (there is also a QWERTY version of this design which is smaller than a credit card). The Fastap™ keyboard uses a "hills and valleys" approach in which the hills are alphabetic characters and the valleys are numeric characters. Clicking a hill produces that key (i.e. alphabetic letter). However, the valleys are effectively just a rubber plate connecting the four surrounding hills and so when a valley is pressed, you are effectively pressing all four of the surrounding hill simultaneously which results in the relevant number being pressed. As expected, pressing a valley is a bit harder than pressing a hill, but as the valleys are larger in size, the effect is both pleasing to the eye and makes for a surprisingly easy-to-use entry method [24].

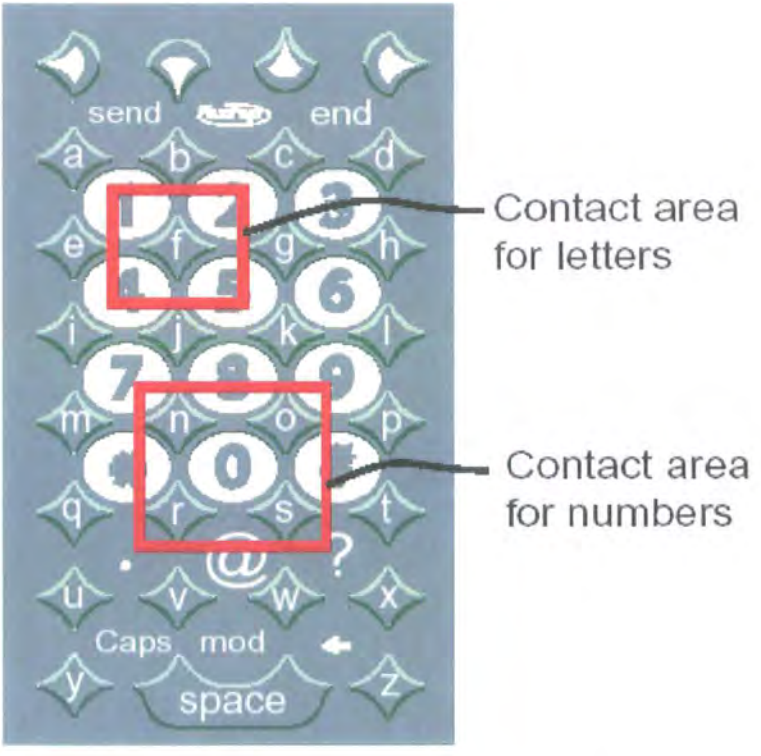


Figure 2-8 Digit's Fastap™ Keypad

The Fastap™ keyboard is a very interesting concept but there may be problems if the keypad itself is made significantly smaller than it is in Figure 2-8 in which case the usability will rapidly be reduced due to the size of the buttons. However at present

Fastap looks like a very promising technology for improving text entry in mobile phones. A mobile phone with a Fastap™ keypad is shown in Figure 2-9.

Figure 2-9 Mobile Phone with Fastap™ Keypad[5]

2.4.8 Voice Recognition

Voice recognition provides the only solution for many "hands free" applications. Whilst it provides a good solution for short messages it is not suitable for long messages as it is very tedious to edit text using voice commands not to mention the (high) 50% drop rate. Another major problem is that text entry by dictation compromises the users privacy; by definition the user has to read out what they are trying to write making voice recognition an unsuitable technology for use in public places. Voice recognition also works poorly in areas with background noise. A recent study showed that the rate of text entry in corrected words per minute for a voice recognition system is still only 13.6 wpm compared to 32.5 for entry using a keyboard [25]. It was also noted that users found it significantly harder to "talk and think" than to "write and think".

Despite the observations made above, Intel's VP Howard Bubb claims that "Speech will become the primary interface, especially in mobile computing" as "the (computer's) processors are becoming tailored to human interaction" [26].

Presently voice recognition does not provide a viable alternative text entry method, but some of its failings can be rectified by further development such as the high drop rate

[5] Reproduced with permission from Digit Wireless.

of existing systems. Other problems with voice recognition such as the lack of privacy caused by reading out the text in public places are inherent and cannot be solved.

2.4.9 Evaluation of Existing text Entry Methods

The text entry methods summarised above demonstrate that there is a need to find an alternative method for entering text into mobile devices such as mobile phones, PDA's and palm top computers if the usability of these devices is to be improved and their uses extended. Although many new methods of entering text have been covered there is still not a good solution. There is a fundamental dilemma with designing a new text entry method for mobile devices; the system needs to be both new and revolutionary whilst also being very easy for a "walk-up" user to learn in minimal time.

It has been shown that the use of the QWERTY layout in soft keyboards is far from optimal as the QWERTY layout was designed for two handed operation (as in typing) rather than pointing with a single stylus. Other layouts such as Opti, Atomik and the Metropolis keyboard result in a significant improvement in performance. These keyboards perform at rates of around 40wpm – approaching the 50wpm performance of a physical (full-size) QWERTY keyboard. The performance of a dynamic layout such as FOCL was low due to the confusion caused by changing the key layout.

Of the existing text entry methods in mobile phones, triple tap and T9, T9 performs better with 41 – 46 wpm compared to 21 - 27 wpm for triple tap. However, it should be noted that this increased performance is for "expert" users and must be set against the time taken for new users to learn how to use the technology. Digit's Fastap™ looks very promising, offering enhanced text entry – it would be interesting to see how it performs in comparison to T9.

Voice recognition remains an unattractive option for mobile text entry due to the high drop rate and the lack of privacy. Handwriting recognition is widely used in PDA's although users experience quite a lot of frustration when learning to use the system. The performance of Graffiti™ is not particularly impressive at around 15-20wpm.

In summary, for a device that is primarily a 'phone, T9 or possibly Fastap™ provide the best method for entering text. For PDA type devices, the best solution is a non-QWERTY optimised soft keypad such as Opti, Atomik or Metropolis. It is interesting to note that these technologies are not widely used in existing devices, primarily due to the users reluctance to learn or use a new technology.

Text entry remains the single most limiting factor in the design and use of mobile devices as shown by the relatively slow text entry rates present in existing devices. Bluetooth may provide an alternative solution by means of a single portable text entry method that can be used to enter text into all devices via Bluetooth.

2.5 CHAPTER SUMMARY

This chapter has discussed the need for the PAN Gateway device and its concept and has then evaluated the various text entry methods for mobile devices. Finally the usability of Bluetooth in existing mobile devices was investigated. The PAN Gateway is a device that allows users to create a network of personal devices (using Bluetooth) such as a PDA, laptop, camera, headset and link them to external networks using a GSM/GPRS modem. The PAN Gateway is advantageous for network operators as it reduces the cost of the basic unit that they provide to customers.

Text entry methods were discussed and it was determined that the QWERTY layout is unsuitable for use in soft keyboards where a single stylus is used to "press" the key. The best text entry method for a PDA type device was optimised non-QWERTY soft keyboards giving text entry rates of around 40 wpm. For a 'phone type device the best method was T9, giving up to 56 wpm or potentially Digit's Fastap™ layout.

The short-range wireless technologies that could be used in the PAN Gateway to provide local connectivity will be discussed in more detail in Chapter 3.

CHAPTER 3 TECHNOLOGIES FOR SHORT RANGE WIRELESS COMMUNICATION

There are various short-range wireless technologies that could have been used to connect the mobile devices to form a Personal Area Network (PAN). The various technologies are discussed in this Chapter to demonstrate why Bluetooth was chosen as the best technology for use in the PAN Gateway. The coexistence of RF systems using the 2.4Ghz ISM band is discussed in addition to a brief section on the usability of Bluetooth in existing devices.

The technologies considered that could be used to achieve short-range wireless communication fall into two groups, those that use infrared light and those that use radio frequency (RF) signals. In the Infrared category there is IrDa (Infrared Data Association), in the RF category there is HomeRF, Bluetooth, IEEE802.11b (Wi-Fi™) and IEEE802.15 – these will now be considered in more detail: -

3.1 IrDA

IrDA-Data is the standard that is referred to by IrDA in this report. IrDA is a low power, low cost, cable replacement technology used for short range Line-of-sight Communication. "IrDA is a point-to-point, narrow angle (30° cone), ad-hoc data transmission standard designed to operate over a distance of 0 to 1 metre and has data speeds of 9.6kbps to 16 Mbps" [27].

In the year 2000 IrDA had an installed base of over 150 million units with an annual growth rate of 40%. IrDA is widely available in portable devices such as PC's, notebooks, peripherals, mobile telephones, PDA's and embedded systems. IrDA has been universally adopted and accepted worldwide.

The main limiting factor for IrDA is that it a line-of-sight technology and thus requires the two devices to be aligned throughout the communication. This limitation makes IrDA unsuitable for use in the PAN Gateway.

3.2 TECHNOLOGIES USING THE 2.4 GHZ ISM BAND

Bluetooth, HomeRF, 802.11b and 802.15.3 all use the (globally) license free 2.4 GHz Industrial Scientific Medical (I.S.M) band and support wireless networking. Yeadon [28] claims that the market for wireless connectivity is one of the fastest growing in history and that this is the case because of applications in the communications infrastructure, industry and business as well as the consumer market.

To use the 2.4GHz ISM band, the FCC[6] requires devices to use Spread Spectrum technologies.

3.2.1 Health Issues

Microwave ovens operate at around 2.4GHz as this is the frequency that is most effective for heating water molecules. Intuitively, a device that broadcasts at 2.4GHz would seem likely to be a danger to human health as humans are made up primarily of water.

Dempsey, in his paper "The Physiological effects of 2.4GHz Frequency Hopping Radios" [29] states that are three perspectives from which this needs to be investigated:-

1. Traditional Thermal effects – i.e. the body being heated.
2. Cellular Interactions – i.e. cell mutations.
3. Effects of the 2.4 GHz radio on a piece of medical equipment (which could be life critical).

Dempsey concludes that there are no adverse biological effects that are caused by the 2.4 GHz radio. In terms of Cellular effects, to date there appears to be no credible evidence to suggest that there are any adverse effects caused by using a 2.4 GHz radio. However the research to date has not proven that there is a fundamental reason why this must be the case.

With regard to interference problems between 2.4GHz Frequency Hopping Spread Spectrum systems and medical equipment he states that historically, very few medical device problems that have been reported have been caused by a radio; only 0.007% of the total problems reported between 1979 and 1995. It should also be noted that a 2.4

[6] FCC – Federal Communications Commission

GHz radio produces field strengths that are only 20% of the typical minimum radiated susceptibility level of most medical equipment and that these field strengths are significantly lower than other radios that historically operate in the hospital environment. At this point it is believed that the potential benefits of using a 2.4 GHz FHSS radio in a medical environment greatly outweigh the potential risks.

3.2.2 Spread Spectrum Modulation (SSM)

"Spread Spectrum is a means of transmission in which the data of interest occupies a bandwidth in excess of the minimum bandwidth necessary to send the data" [30]. SSM is a digital coding technique in which a narrowband signal is spread over a spectrum of frequencies. The coding operation increases the number of bits transmitted and the bandwidth used. There are two main types of Spread Spectrum Modulation: -

1. DSSS – Direct Sequence Spread Spectrum.
2. FHSS – Frequency Hopping Spread Spectrum
    a) Slow Frequency Hopping.
    b) Fast Frequency Hopping.

Direct Sequence Spread Spectrum

In DSSS the radio energy is spread across a larger bandwidth than is necessary by dividing each data bit into sub bits. The higher modulation rate is achieved by multiplying the digital sequence with a pseudorandom noise sequence known as a chip sequence. For example, if the chip sequence is 10 and it is applied to a signal carrying data at 300kbps, the resultant signal will have 10 times the original sequence's bandwidth. The spreading is achieved using a specific code thus creating a unique spectrum that only a receiver using the same code can collapse into its original form. The spectrum of a DSSS signal appears to be noise, making it very difficult to detect. 802.11b uses DSSS.

Frequency Hopping Spread Spectrum

In FHSS the transmitter jumps from one frequency to another at a specific hop rate. The order in which the frequencies are used is determined by the pseudo random hop sequence used. The FCC mandates that FHSS systems spend no more than 0.4 seconds on any one channel each 30 seconds and that they must hop through at least 75 channels in the 2.4 GHz band.

The use of a frequency hopping spread spectrum method of modulation improves immunity to interference from other devices. In FHSS systems, data is transmitted for a very short time (determined by the hopping rate) on a particular frequency before hopping to the next frequency. Interference typically occurs at a single frequency thus only a few of the frequencies that are used by the FHSS system are likely to be impaired by interference, making FHSS resistant to interference. FHSS systems can be susceptible to noise during any one hop but typically can achieve transmission during other hops within the wide band. There is also the potential for adaptive frequency hopping in which the system determines which frequencies are being degraded by interference and the system effectively "hops around" those frequencies [31].

Transmission using Spread Spectrum Frequency Hopping typically appear to be background noise unless the systems are synchronised and the receiving station knows the pseudo random hopping sequence, making the signal resistant to detection, interference and jamming.

In slow hopping systems, several symbols are transmitted on each hop; in fast hopping systems, the carrier frequency hops several times during the transmission of one symbol. Bluetooth uses slow hopping FHSS.

3.2.3 IEEE802.11b

The 802.11b specification was written by the IEEE[7] 802.11 working group. 802.11b is designed for use primarily as a wireless Ethernet and allows transmission speeds of 11 million bits per second, making it faster than conventional wired LANS (although upgrades to wired LANS now allow for data rates of 100Mbps). It uses Direct Sequence Spread Spectrum in the 2.4 GHz ISM band and is designed to be implemented by IT professionals in an office environment. 802.11b is more expensive to implement and uses more power (although this does give greater range). Security is a major issue with 802.11b; it has been widely reported in the press that many large companies have failed to implement sufficient security measures on their wireless networks allowing hackers to access sensitive information and to use their web connection for free.

[7] IEEE – Institute of Electrical and Electronic Engineers

Cisco, Lucent, Apple and 3Com all have large stakes in this technology as part of a special interest group called WECA (Wireless Ethernet Compatibility Alliance); they are already pushing for it to become the wireless Ethernet standard. 802.11b is also known as Wi-Fi™.

3.2.4 HomeRF

The HomeRF™ Working Group (HRFWG) was set up in March 1998 to create an open industry specification for wireless digital communication between PCs and consumer electronic devices in and around the home. It was also set-up to act as a forum for the encouragement and support of home wireless networking [2]. HomeRF uses Frequency Hopping Spread Spectrum in the 2.4 GHz ISM band and is incompatible with 802.11b. HomeRf is specifically designed for use in the home and is easy to set-up as it was designed for consumer use. HomeRf has many advantages over 802.11b for use in the home, such as easier configuration, but there seem to be very few HomeRf products available to consumers.

HomeRF also has substantial Industry backing, including Intel, Microsoft, Motorola, Proxim and Siemens amongst others in the HomeRF working group. Recently the FCC approved an upgrade from 2Mbps to 10 Mbps making it a viable challenger to 802.11b for office use.

3.2.5 Bluetooth

"Bluetooth is a low cost, low power, short-range radio technology, originally developed as a cable replacement to connect devices such as mobile 'phone handsets, headsets, and portable computers" [2]. This can enable a very powerful ubiquitous computing platform where each and every device is connected to the network [32]. Bluetooth uses Frequency Hopping Spread Spectrum in the 2.4 GHz ISM band; with a hopping period of 62μS. Bluetooth has a series of profiles that describe how particular applications can be achieved, including which parts of the core Bluetooth protocol should be used to support the profile. In order for a device to support certain functions, the relevant Bluetooth profile must be supported; for example without a Bluetooth implementation incorporating the headset profile, it will not be possible to use a headset. The Bluetooth profiles include the following [2]: -

Generic Access Profile – this is the core Bluetooth profile. Its purpose is to ensure that Bluetooth devices can all establish a baseband link.

Dial Up Networking – provides a dialup data connection. For example it allows a laptop to be used to check email via a mobile phone.

Headset Profile – defines the facilities required to make and receive voice calls from a headset to a mobile phone.

LAN Access Profile – allows Bluetooth enabled devices to connect to a fixed network via a Bluetooth link to LAN access point.

Synchronisation Profile – provides a standard way for personal information to be synchronised between Bluetooth enabled personal devices such as PDA's, laptops and cell phones.

Personal Area Network Profile – provides support for ad-hoc networks in the form of full TCP/IP networking. PAN uses BNEP, the Bluetooth Network Encapsulation Protocol, to transport common networking protocols over wireless links. Essentially the BNEP allows Bluetooth to carry IP packets allowing the PAN profile to offer a wireless LAN function based on IP. For example the PAN profile is used to link computers to peripherals and 'phones to PDA's and headsets, providing the ad-hoc networking capability. The PAN profile is fundamental to the PAN Gateway.

Bluetooth has a different usage scenario to 802.11b and HomeRF as it was originally intended as a cable replacement technology and not as a Wireless LAN. Wireless LAN functions can be achieved through use of the LAN Access profile. Bluetooth presently only supports data rates of 1Mbps and is designed primarily to connect mobile devices such as laptop pc's, mobile 'phones, PDA's and headsets wirelessly.

Bluetooth is widely backed by major industry players including 3Com, Motorola, Toshiba, Nokia, Microsoft, Ericsson Sony, Microsoft, Lucent, IBM, Intel and over 2000 other companies. Bluetooth is expected to be able to replace six to eight ports on a computer with virtual ports. Bluetooth has received much press coverage and hype but has taken longer to rollout than expected.

3.2.5.1 Key Features of Bluetooth

- Uses globally license free Industrial Scientific Medical (I.S.M) Band at 2.4GHz.
- Omni-directional, not limited by a small transmission angle.
- Not limited by line-of-sight (LoS).
- Robust.
- Low Complexity Hardware.
- Low Power (Active power is 0.1 Watts). Uses power saving modes.
- Allows transmission rates up to 1Mbps, although real throughput (without overhead) is 721kbps.

- Low Cost ($5 basic hardware cost in the long term).
- High security, uses pseudorandom hop sequence.
- Low power (range ~ 10m) and high power (range ~ 100m) versions

Bluetooth has been designed to operate in (electronically) noisy environments and uses spread spectrum frequency hopping to achieve this. Bluetooth typically hops between 79 different frequencies at a rate of 1600 hops/second. If the transmission encounters interference, it waits for the next frequency hop and re-transmits on a new frequency. The available channel in the 2.4 GHz ISM band is usually divided into 79 slots (except in Spain, France and Japan, where there are only 23 slots available). Each slot corresponds to an RF channel. Each channel is displaced by 1MHz in the range 2.402GHz – 2.480GHz, with additional guard bands at either end of the spectrum. Bluetooth systems use shorter packets and hops faster than any other system in the 2.4Ghz band.

3.2.5.2 Architecture/Topology

The Bluetooth topology is best described as a multiple Piconet structure. A Piconet is a kind of miniature LAN formed by a device configured as a Master, which 'owns the Piconet' and between one and seven devices that always act as Slaves to this Master" [2].

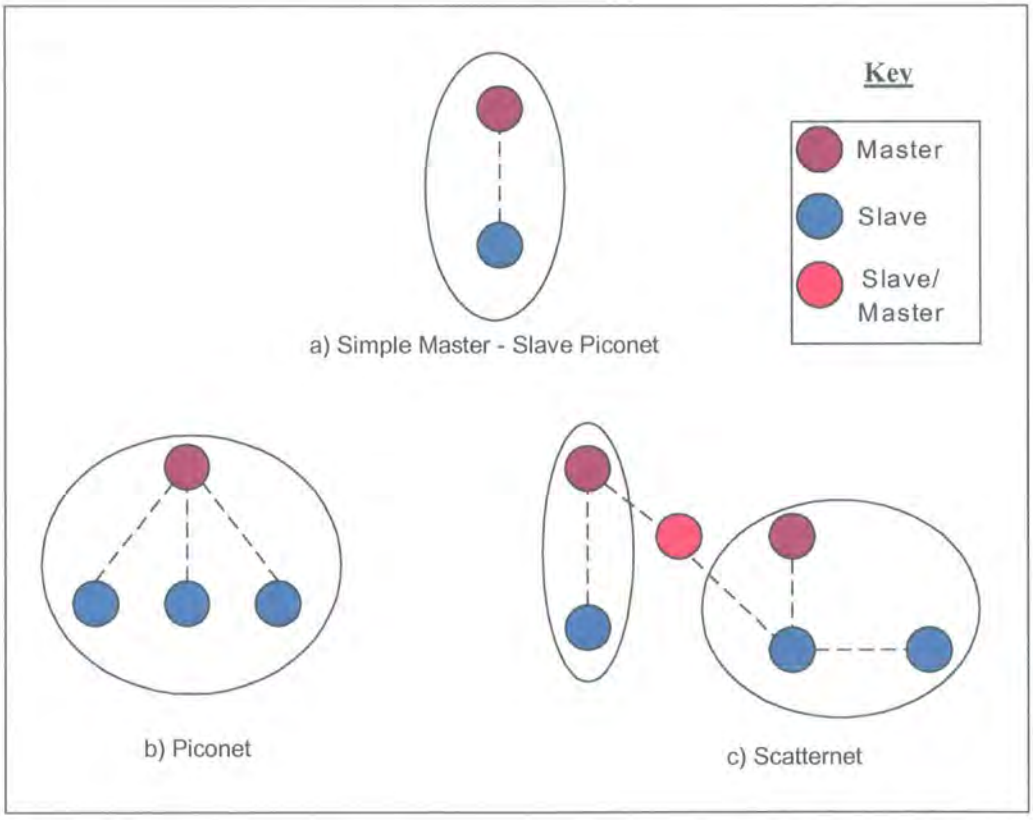


Figure 3-1 Bluetooth Topology Diagrams

Figure 3-1 shows three Bluetooth architectures, Figure 3-1a) Simple Master-Slave Piconet, Figure 3-1b) Piconet, Figure 3-1c) Scatternet.

The simplest system consists of a Master and a single Slave (see Figure 3-1a) [1]. Master's can control up to seven Slaves in a piconet, (see Figure 3-1b) [31] [33]. Several piconets can be established and linked together ad hoc to form a Scatternet (see Figure 3-1c) Scatternet).

Devices in a Piconet share the same channel in which the Master is defined as the device that initiates the call, although the specification does support Master-Slave role swapping. The pseudorandom hop sequence is generated from the device address of the Master and the phase of its clock. The Masters' clock is used as the Piconets' clock to synchronise all of the devices. Although a device can be in more than one Piconet by means of time division multiplexing, it can only be the master of one Piconet. Piconets are uncoordinated with each other and frequency hopping occurs independently; synchronisation of Piconets is not supported.

Communication in a Piconet is organised such that the master polls each slave. A slave is only allowed to transmit after the master has polled it, it then transmits immediately after the poll in the slave–to-master time slot [1].

Work is being carried out on allowing devices to support multiple profiles simultaneously to meet user expectation. For example, it is necessary for a mobile phone to be linked to both a PDA using the dial up networking profile and a headset using the headset profile simultaneously. The above scenario would be used to call a number from the users PDA whilst allowing the voice call to be handled via the headset [33].

3.2.6 IEEE 802.15

The IEEE is set to release a new standard, the 802.15.3 specification, at the end of this year that will heavily outgun Bluetooth in terms of data rate; providing short-range rates of 20Mbps in comparison to Bluetooth's 1Mbps. However, it is anticipated that as 802.15.3 will be backwards compatible with Bluetooth, that it will stimulate the market for Bluetooth as it'll be a few years before any 802.15.3 products are available, whereas as Bluetooth products are available now.

A second 802.15 specification has also been drafted – offering data rates of 55Mbps throughput, suitable for high-end video distribution within a home. The specification was created because "no other wireless standards can simultaneously distribute three different digital video streams, one internet [connection] and three 'phones, and one CD audio stream perfectly" said Bob Heile, 802.15 working group chairman [34].

3.3 Coexistence of Bluetooth and IrDA

Both Bluetooth and IrDA are cable replacement technologies, however they are not in direct competition with each other as each technology has its own strengths and weaknesses. For example, IrDA is directional and can only be used for line-of-sight connections and therefore has some built in security features, whereas Bluetooth can be used over longer distances and is omni-directional.

Both IrDA and Bluetooth consider data exchange to be a fundamental function and use the OBEX upper layer protocol. By using the same upper layer protocol it is possible for a single application to run over Bluetooth and IrDA. It is the intent of both Bluetooth and IrDA to utilise the same data exchange applications where appropriate. The presence of both technologies allows the user to select the most appropriate method of communication based on the present situation [27].

Although the technologies can be found in similar devices, their applications are inherently different. IrDA is suitable for applications where data transmission takes place at high speeds over a closely proximate line of sight path. Meanwhile Bluetooth is more suitable for situations where a line of sight connection is not possible or the two devices to be connected are not stationary [35].

3.4 Coexistence in the 2.4 GHz Spectrum

With many types of devices using different specifications sharing the same area of the spectrum (the 2.4 GHz ISM band), questions must be raised with regard to whether these different technologies can coexist. Does an 802.11b device interfere with a Bluetooth connection? Is the data rate or the range of the devices reduced? In a recent study Cordeiro and Agrawal [36] concluded that Bluetooth devices themselves are likely to be interferers to the Bluetooth technology in the near future. They also observed that Bluetooth has a very high overhead in the current Bluetooth Piconet switching procedure.

The diagram below, Figure 3-2, shows the overlapping usage of the spectrum when 802.11b DS (Direct Sequence), 802.11 FH (Frequency Hopping) and Bluetooth are used [37]. HomeRF also uses this spectrum but has not been shown on this diagram.

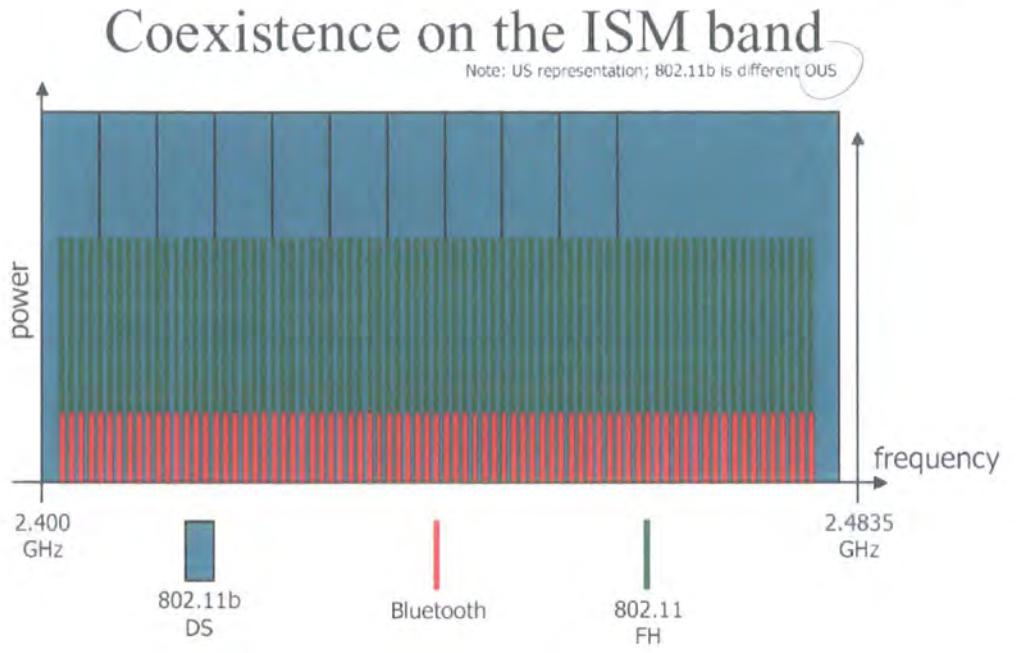


Figure 3-2 Diagram of spectrum usage in the ISM band

Co-channel interference occurs when a Bluetooth transmitter hops into the occupied channel of an 802.11 Direct Sequence network [38]. The diagram gives the impression that these technologies cannot coexist; however this is not the case due to the nature of the technologies. At a particular time each transmitter and receiver is tuned to a particular frequency and provided that the radios are narrow and clean (i.e. do not overlap) and excepting the case when they are on the same frequency, coexistence is possible. How often two stations will be using the same frequency will depend on the number of stations, the number of frequencies available and the time spent at each frequency.

In a report for the IEEE [38], Zyren concludes that the interference from a Bluetooth piconet is a localised effect and the range at which interference from a Bluetooth piconet seriously degrades the 802.11 network is dependent on the direction of data flow, local propagation conditions, the 802.11 data rate and the Bluetooth piconet utilisation. Shorter packet sizes allowed by the 5.5 Mbps and 11 Mbps 802.11b systems improved the error free throughput rate and a fragmentation threshold of 750 bytes provides good throughput under heavily loaded conditions, without having a severe throughput penalty when no interference is present.

In another report [39] Zyren concludes that 802.11 susceptibility to interference increases as a function of range from the DSSS wireless node to the DSSS Access Point and that 802.11 DSSS shows a graceful degradation in the presence of significant levels of Bluetooth Interference. At the 2001 Bluetooth developers conference in San Francisco, Dell Computer Corp acknowledged that they are intending to put Bluetooth and 802.11b in the same PC card, which gives rise to some interesting interference issues. However in one study [37], interference between the two frequency hopping technologies (at reasonable data rates), 802.11 and Bluetooth, was found to be minimal at less than 10%.

Interference between the Frequency Hopping (FHSS) and Direct Sequence (DSSS) technologies is a rather different story. Direct Sequence technologies provide a large quantity of interference as shown by their spectrum usage (see Figure 3-2). In the US, DSSS technologies use three 22MHz bands simultaneously when heavily loaded, interfering with FH devices. However the degradation of FHSS devices on DSSS devices is even worse. In a report for the IEEE [40], it was shown that Bluetooth can impact 802.11 DSSS significantly, particularly on large packets. In addition most 802.11 mechanisms for responding to poor channel quality either have no impact or make things worse. It was also found that fragmentation can help, mainly at lower data rates and high picocell utilisations.

Proxim, Mobilian [41] and Texas Instruments [42] have all performed similar tests on the effect of Bluetooth Interference on an 802.11 DSSS system and have produced consistent results. A Bluetooth Interferer was placed 12 feet away from a 802.11 or a HomeRF node and the resulting impact on data throughput on the victim node was measured. The 802.11 node located 12 feet from its Access Point degraded about 25%, whilst a HomeRF node only degraded 10%. HomeRF incorporates "hopset adaptation" and "subframe hopping with retries" in the upper layers in order to increase immunity to 2.4GHz interference [43].

In an Ericsson report for the Bluetooth SIG working group [44] based on a typical office environment and measuring the effect of a 20dBm 802.11 DS WLAN on a 0dBm Bluetooth system, it was found that if the Bluetooth connection was less than 2m long the probability of disturbance on a Bluetooth voice link was less than 1%. If the Bluetooth link increases to 8m, probability of disturbance rises to 8%. For a 10m Bluetooth data link a throughput degradation of more than 10% occurs with a probability of 24%. "Due to the limited frequency overlap of the WLAN and Bluetooth

systems, the throughput reduction in the Bluetooth system can never exceed 22%" [44].

However in the Texas Instruments test [42], where degradation on a Bluetooth system located 10cm from an 802.11 system was measured it was shown that the degradation of the Bluetooth System throughput was at least 40%; when the spacing was increased to 10m the degradation dropped to 10%.

During testing Mobilian demonstrated that lowering the transmission power does not change the basic shape of the Wi-Fi™ performance degradation curve, but it shifts the curve to the right, increasing the range over any which any given throughput is available [41].

However, not all reports agree on this topic. A study conducted by the "Pennsylvania State University's Applied Research Laboratory" and Wireless Infotech Services found that "Bluetooth and 802.11b wireless local area networks can co-exist without interfering with each other's operation" [45]. The groups stated that the range of both Bluetooth devices and 802.11b devices were also found not to be affected by the presence of a device using the other technology, no matter how close they were. Bluetooth's range was found to be about 64 feet and 802.11b range was found to be around 284 feet. The groups agreed that more tests were required to evaluate performance in different field conditions. No tests were carried out to evaluate the effect of presence of a device using the other technology on the speed of the transmission.

3.4.1 Solutions to the ISM Band Co-existence Dilemma

Various solutions to coexistence issues are being developed. Adaptive Frequency Hopping (AFH) is the leading non-collaborative technique for minimising interference problems between 802.11 and Bluetooth.

AFH works by reducing the number of Bluetooth channels a Bluetooth device uses, thus leaving channels free for other devices to use. Without AFH, Bluetooth uses 79 of the available 83 channel; with AFH, it is likely that only 15 channels will be used leaving up to 68 free. AFH is suitable for use in devices that do not have Bluetooth and 802.11 co-located, in these devices it is necessary to provide a further solution [46].

Bandspeed, Inc. and Open Interface, North America has announced a new product that utilises Adaptive Frequency Hopping (AFH) to allow 802.11b and Bluetooth to co-exist. The solution uses Bandspeed's chipset with Open Interface's BlueMagic protocol stack and is designed to provide a coexistence solution for hardware manufacturers and OEM's. "BlueMagi AFH is backwards compatible with Open Interface's Bluetooth Spec version 1.1 and works with existing Bluetooth wireless devices" [47].

Similarly, U.K. based Red-M have produced "Genos"; a "cutting edge software solution for creating a stable, multi-technology network environment that enables both Bluetooth and 802.11 enabled devices to co-exist and interoperate successfully" [48]. AFH significantly reduces the problems caused by interference in the 2.4GHz ISM band.

3.5 Evaluation of Wireless Technologies for the PAN Gateway

Various possible technologies that could be used to provide the local connectivity required for the PAN Gateway have been discussed in Sections 3.1 to 3.4. It was concluded that IrDA was unsuitable, as it requires a line-of-sight between the devices to communicate and therefore could not provide a link between a laptop and a PAN Gateway in a briefcase.

Essentially technologies that use the 2.4GHz ISM band were considered. Bluetooth is a low power FHSS system, designed as a cable replacement technology for mobile devices giving a throughput of 1Mbps. HomeRf is designed as a home networking technology presently offering data rates of 2Mbps using FHSS. 802.11b provides wireless Ethernet connectivity using DSSS at data rates of 11Mbps. Of these, Bluetooth is the most suitable technology for use in the PAN Gateway, as it requires the least power and was designed as a cable replacement technology for small mobile devices with limited resources. With regard to interference issues between the different technologies that use the 2.4GHz ISM band, it was concluded that 802.11b is the most seriously affected and that although Bluetooth immunity to interference was not as good as HomeRf's it could be significantly improved through the use of Adaptive Frequency Hopping (AFH).

3.6 Usability of Bluetooth in Mobile 'Phones

A number of mobile 'phones (Nokia 6210, Ericsson R520m and T39m, Sony Ericsson T68i) were used in informal usability tests to determine if there were any obvious shortfalls in the usability of Bluetooth in existing mobile 'phones.

Bluetooth was determined to be very useful once it had been set up for sending emails whilst on the move (e.g. in a train) without having to try to line up the IrDA ports and hold them steady. The use of headsets (where applicable) was very briefly tested and concluded to be a positive experience. However, the pairing procedure on all the 'phones tested seriously diminished the user's "Out of Box" experience. The pairing procedure is both complex and unintuitive in addition to having shortfalls in terms of security.

To carry out Bluetooth Pairing the following steps must be completed: -

1. Perform a device discovery to determine what other Bluetooth devices are in the area.
2. Select the Bluetooth device you want to connect to from the list.
3. Enter the security PINs for both devices.

This procedure has many problems, which have been outlined below.

- The device discovery may return a long list of Bluetooth Enabled devices, making it difficult to select the correct device to connect to, e.g. there may be four devices named "John's laptop".
- How to exchange PIN numbers? Users are likely to tell the owner of the other device their device's PIN number, making it easy to "overhear" the PIN.
- In a Bluetooth SIG Security White Paper [49] it was stated that "we also recommend that the user be in a "private area", before using the pairing procedure from the Bluetooth Baseband Specification". I.e. the Security provided by the baseband pairing mechanism is not sufficient.
- The process is non intuitive and different for each device.

In summary, there are significant issues relating to the usability and security of the Bluetooth pairing procedure with respect to the average user's "Out of Box" experience.

3.7 Future of Bluetooth

The long-term future of Bluetooth is very promising, primarily due to the potential impact of the technology. The freely available specification has resulted in a large number of companies investing heavily into researching Bluetooth, giving the technology wide industry support. Presently there are more than 2000 members of the Bluetooth Special Interest Group (SIG).

Despite a slow start and negative publicity in the last year many reports are now suggesting that a Bluetooth revolution is just around the corner. At the 2001 Bluetooth developers conference in San Francisco, Microsoft announced that it planned to provide native support for the Bluetooth standard in future versions of Windows XP [50]. Predictions about the future use of Bluetooth range from 1.16 billion Bluetooth chipsets being produced in 2005 (Jack Quinn, Micrologic Research) to 1.4 billion devices incorporating Bluetooth technology being manufactured in 2005 (Joyce Putscher, Cahners In-Stat). A recent report by the ARC group has suggested that market penetration of Bluetooth in mobile 'phones will reach 75% by 2006. Market indicators show that there will be over 1 billion mobile 'phone subscribers by 2006 which implies healthy opportunities for Bluetooth.

It was also noted that the pending Bluetooth products would be less ambitious than those originally envisioned. Bluetooth's focus has presently shifted away from the ad-hoc networking devices as the access points and PC cards implementing the 802.11b specification fulfils this role. Presently the focus is on small, portable low power devices such as PDA's and mobile 'phones where the speed or range of the connection is less important than the requirement for low power consumption. Although much of the functionality that was seen as Bluetooth's domain has been lost to 802.11b, such as wireless Internet access in café's such as Starbuck's, it should be noted that the two technologies have many different features and do not compete directly against each other.

The Bluetooth SIG is presently working on the Bluetooth 2.0 specification that will increase data rates to 12Mbps (compared to 802.11b's 11Mbps). However, Cambridge Silicon Radio's marketing Vice President, Eric Jansen believes that "the real goal is to get products embodying the 1.1 specification into production" and that the 2.0 specification is "over hyped" [50].

One major advantage that Bluetooth has in the long term is its low cost and small form factor allowing it to be cheaply and easily embedded into many products. Ericsson Component's technical manager, Lars Nord stated that Ericsson's latest radio, the ERC41 requires only eleven external components, is half the cost of the previous module by integrating a variety of on chip components and consumes a maximum of 27 milliamps.

In summary, it seems that Bluetooth is a very promising technology which has taken longer than anticipated to reach the market. The market for Bluetooth products has

shifted away from LAN type networking and towards low cost, low power, small form factor embedded systems allowing ad hoc links between mobile devices. A higher speed specification is also being worked on.

3.8 CHAPTER SUMMARY

In this chapter the various technologies that could have been used to provide the local connectivity for the PAN Gateway were discussed. IrDA was determined to be unsuitable for the PAN Gateway due to requiring a Line-of-Sight between the devices to be connected. It was concluded that Bluetooth was the best short-range wireless technology to use in the PAN Gateway as it is low power, low cost, omni-directional and designed for use in mobile devices.

The interference issues surrounding the use of 2.4GHz wireless technologies were discussed and it was concluded that although Bluetooth does suffer from interference from other wireless technologies (such as HomeRf and 802.11b) the effects are not as significant as those on 802.11b and can be reduced through the use of Adaptive Frequency Hopping (AFH). There appear to be no significant health concerns with the use of 2.4GHz spread spectrum wireless technologies. In the process of evaluating the usability and security of Bluetooth in existing Mobile Devices, a significant problem with usability and security was discovered in the Bluetooth Pairing procedure. In addition to not being intuitive the Bluetooth SIG had recommended against the pairing of devices in public places. This problem is one that will need to be addressed in Bluetooth implementations in mobile devices including the PAN Gateway.

In summary, Bluetooth is the most suitable wireless technology for use within the PAN Gateway to connect to local devices. The Bluetooth topology is based on a multiple Piconet structure and PAN's are supported by the Bluetooth Personal Area Network profile. Bluetooth itself is a relatively new technology that is expected to be in widespread use in the next 4 years.

In Chapter 4 the requirements of the PAN Gateway with respect to both functionality and usability will be discussed, taking on board the lessons learnt from Chapter 2 and Chapter 3.

CHAPTER 4 REQUIREMENTS OF THE PAN GATEWAY

In this chapter the information gathered in Chapter 2 and Chapter 3 is used to consider the requirements of the PAN Gateway device and to determine the best user interface for use in the PAN Gateway. A new concept for improving the usability of Bluetooth pairing is introduced.

4.1 AIMS

The aim of the PAN Gateway is to increase both the flexibility and functionality of the mobile 'phone, whilst reducing the cost to the operator for the basic handset. Presently, the cost of the mobile 'phone provided to users is ever increasing as manufacturers integrate more and more functions into their phones, but the price users are prepared to pay for their 'phone remains minimal with network operators picking up most of the cost of the 'phone.

This initial outlay (to cover the cost of the phone) by the network operators is recovered over subsequent months through line rental and call charges. Meanwhile the cost of voice services to the consumer is being driven down by competition for customers and increased usage and so network operators are searching for ways to get customers to use their phones more and more, such as text messaging services (SMS) and the new Multi-media Messaging Service (MMS).

The PAN Gateway is a device consisting of a Bluetooth modem and a GSM/GPRS modem; it would be used as the part of a Personal Area Network that provides access to external networks via the GSM/GPRS modem. For example, the PAN Gateway would allow email to be checked from a PDA or laptop using the Bluetooth Dialup Networking profile, or for a voice call to be made or answered using a headset. The PAN Gateway should also allow calls to be routed through land based telephone lines where possible, significantly increasing the users perception of functionality. This is similar to the concept of having mini GSM aerials located within offices so that if users call from within the office the mobile network is used (assuming that there is sufficient capacity) but it is billed to the company at landline rates.

It is imperative that the pairing procedure required for using Bluetooth is simple. Initial barriers that occur during registration prevent users from using new technology – until you know the value of a new technology you will not spend time getting over the initial barriers. For example registering your WAP handset to a service portal can be difficult

enough to prevent users from accessing WAP services; only when configuration is automatically downloaded to the handset is this barrier removed. Similarly around 40% of people who own a 'phone with T9 Predictive text do not use it because they cannot get to grips with how it works. Another example is online banking, if registration takes more than a few minutes then the chances are that the service will be unsuccessful due to these initial barriers.

4.2 CONCEPTS

It is important to consider the exact circumstances in which the PAN gateway will be used in order to maximise the usability of the device. Features that make a product useful vary according to the product and usage scenario; for example on a PC, users don't mind waiting a few seconds while an application loads because they plan to use the application for a certain amount of time, whereas on a PDA people want instant access to information, i.e. speed is critical.

The PAN Gateway enables a modular design of mobile functions which allows the operator to provide (subsidise) the user with a basic set at a reasonable cost to the operator. The basic unit can then be enhanced by purchasing additional modules providing a much wider range of features than are presently available to the user.

There are a number of ways of modularising the PAN Gateway system depending on which features are included in the base unit. Other factors considered include how to link the various modules; "express-on" interfaces such as keyboards (similar to Nokia's "express-on" fascia's)? Plug in modules? A major consideration is how to allow Bluetooth Pairing to occur between two devices such as a headset and base mobile 'phone unit if neither device has a screen or user interface.

The initial concept for the PAN Gateway base unit is to have a small device that can be kept in a pocket, in a brief case or even on the users belt. The base unit will act as a gateway and will contain a GSM/GPRS modem, a Bluetooth Modem, Antenna, Power supply (see Figure 4-1).

Figure 4-1 Block Diagram of the PAN Gateway.

The primary design issues centre on what interface to include in the base unit in order to make it as functional as possible, allowing greatest flexibility at a reasonable cost to the operator. There are various MMI[8] arrangements that can be used which are summarised in the following section. All of the described interfaces could be extended by means of additional Bluetooth enabled modules and these are described in Section 4.2.2.

4.2.1 Possible MMI's for the PAN Gateway

The various MMI concepts for the PAN Gateway are discussed below. The provision of emergency cover, i.e. the ability to call the emergency services with just the PAN Gateway was considered at length and determined to be difficult to implement on the devices with minimal user interface's.

4.2.1.1 Minimal User Interface

The most basic unit has no screen and no keypad. This basic device uses L.E.D's to indicate power levels, network coverage and Bluetooth connection. It has a single button which is a power button. The unit would also contain vibrate and audio alerts to incoming calls, messages and data. This unit is the most compact solution and will also be the cheapest to implement. The device would be small enough to carry on a belt or in a pocket, but would not be able to carry out any functions without other modules connected; it is simply a gateway.

The basic unit could be extended to include a small screen (one alpha numeric row) which would allow more detailed information regarding battery level, network coverage and other status information to be given. These units would not allow calls to be made

[8] MMI – Man Machine Interface

from a headset without the involvement of another module (apart from the base unit) unless voice recognition technology was used.

4.2.1.2 Basic User Interface

This unit would consist of a medium sized screen (four or five alphanumeric rows) and may include a keypad. Without the keypad this device would be very similar to the small screen unit; the enlarged screen (four or five alphanumeric rows) would allow the various power, network, and transmission indicators to be easier for the user to understand but would still not allow much input from the user unless voice recognition technology was used. Potentially the unit could have 'phone book functions – the users 'phone book could be downloaded from a PDA or laptop and then scrolled through with the addition of a rocker button. This unit would not be very user friendly.

If the basic interface was extended to include a keypad, the device is considerably more usable but is very similar to existing Bluetooth enabled 'phones.

4.2.1.3 Advanced User Interface

The devices with an advanced user interface have a medium or large touch screen. The inclusion of a medium touch screen (four or five alphanumeric rows) would allow keypad functionality to be added without the bulk of a keypad. This would allow calls to be made to new numbers. However the unit will still have more limited functionality than a present mobile 'phone due to the touch screen acting as both the screen and keypad. One solution might be to include the symbols indicating power, network coverage and other status information on the top row of the screen which would change to display the number dialled if the touch screen was being used as a keypad for dialling.

The use of an LCD display is likely to reduce the power consumption of the device when compared to the consumption of a few L.E.D's. Similarly the inclusion of a touch screen should not add too much to the cost of the unit as mechanically it is much simpler and it is also smaller. Suitable touch screens might be sourced from Synaptics or 3M.

In order to protect the touch screen it may be necessary for the base unit to have a clamshell type design. In its most integrated form a large touch screen would be incorporated. However the device would be very similar to an integrated phone/PDA.

4.2.2 Other Modules

The following modules could be added via a Bluetooth link to extend the functionality that the MMI's described in Section 4.2.1 would allow: -

Headset - allows calls using voice recognition in the base module.

Keypad - would allow calling to new numbers and enable SMS services.

Bluetooth handset - provides normal mobile 'phone capability. This could potentially be a very slimline device, as it requires no GSM/GPRS modem, just a Bluetooth modem and basic handset functionality. This device would also provide 'phone book functions, SMS functions.

Large Screen and keypad - Designed for web browsing.

It should be noted that any of the modules described above could be incorporated in other devices such as a PDA (large screen and keypad) or alternatively a laptop PC. Similarly the headset could be incorporated into an MP3 playing headset.

4.3 EVALUATION

A major consideration is based on the question "The user has become accustomed to having a mobile 'phone with many functions at little or no cost due to the subsidisation of the unit by operators. What is the minimum level of functionality that the user will now accept?"

4.3.1 Minimal user Interface

The units with minimal user interface have no keypad and either no screen or a screen with a single alphanumeric row. These devices are the simplest and therefore the cheapest to produce; they satisfy the criteria of providing a gateway for communications services with enhanced flexibility, whilst reducing the cost to the operator. However they are limited by having little or no screen, which may make the interface difficult for the user to comprehend and in particular the user may have difficulty in carrying out Bluetooth Pairing.

The operator would almost certainly have to provide a Bluetooth enabled handset to enable voice calls, SMS services and other services that are presently available but as discussed earlier this could be a very slimline unit. The headset and other modules could be made available to the customer at the full price.

4.3.2 *Basic User Interface*

These units have a medium sized screen (four or five alphanumeric rows) and may have a keypad. They have the potential to provide a simpler, more intuitive interface to the user; similar to those provided on mobile 'phones presently.

If the unit did not have a keypad it would limit the functionality of the device, indeed the provision of a medium sized screen with no keypad to use as an interface is questionable. The use of a downloadable 'phone book would reduce reliance on the voice recognition software; dialling of new numbers would still not be facilitated except through the use of a complicated interface or voice recognition and is unlikely to be user friendly.

If a keypad is incorporated into the design the unit is very similar to an existing Bluetooth enabled 'phone and therefore does not adhere to the requirements of a PAN Gateway.

4.3.3 *Advanced User Interface*

The use of a medium or large touch screen has many advantages in terms of usability; one significant advantage of this system is that step-by-step instructions for Bluetooth pairing could be displayed on the screen. However, these devices are essentially the same as the present Bluetooth and GSM enabled PDA's and once again do not adhere to the requirements of a PAN Gateway.

4.4 SELECTION OF OPTIMAL MAN MACHINE INTERFACE

The best MMI that fulfils the criteria of the PAN gateway is the "Minimal User Interface" device with no keypad and no screen. However for this to be user friendly a simple, intuitive Bluetooth Pairing mechanism must be created that does not require instructions to be displayed on screen. This is the best MMI configuration as it allows the base PAN Gateway unit to be used simply as a Gateway. The minimal MMI is the most revolutionary concept and is also the cheapest to manufacture and therefore would reduce the cost to the Network Operator. This PAN Gateway MMI fully supports the concept of user's simply changing over their PAN Gateway for a Gateway that uses a different mobile phone technology when travelling abroad.

However, if no such "simple, intuitive Bluetooth pairing mechanism" can be developed the best solution is the device with the medium touch screen as it would allow step-by-

step instructions to be displayed as well as providing a method for entering text and numbers. In both cases the network operator would need to provide a small Bluetooth enabled handset to provide the functions that users have become accustomed to receiving at little or no cost to themselves. The purchase of Bluetooth enabled headsets, larger screens, keypads and other devices could then be left to the customer.

4.5 Requirements - Usability and the Man Machine Interface

The primary principle behind the design of the MMI of the PAN Gateway is that set up must be intuitive and the device must be user friendly. The user must be able to pair the devices and carry out other vital tasks without needing to refer to a manual.

As discussed above, the best solution to the usability problems caused by the Bluetooth Pairing procedure is to create an alternative intuitive Bluetooth Pairing mechanism that will not require any instructions. However if this is not possible, step-by-step Bluetooth Pairing instructions could be displayed on either the PAN Gateway (using an MMI which has a screen) or alternatively on another terminal that is connected to the PAN.

Displaying the instructions on the PAN Gateway conflicts with the basic need/philosophy of the PAN Gateway, to create a device with a minimal MMI. For the instructions to be easy to read and user friendly, they need to be displayed on a relatively large screen (larger than the present standard mobile screens) whereas one of the aims is to make the device as small as possible.

However to display instructions on other terminals also creates problems; the terminal equipment manufacturers will be relied upon to provide a usable interface to drive the connectivity without reference to manuals. Also what happens if you want to pair a device such as a headset to the PAN gateway – where do the instructions get displayed on a headset?

Another solution may be to use a 3rd display device to display the instructions. This 3rd device could be a PC, a PDA or a viewing terminal provided by the manufacturer. Unfortunately this system relies on the presence of a third device to pair up say a headset and the PAN gateway – which is not necessarily very convenient.

4.6 A NEW CONCEPT FOR AN INTUITIVE BLUETOOTH PAIRING METHOD

The best solution to the usability problem created by the existing Bluetooth Pairing Procedure is to create a pairing mechanism that is so intuitive that no instructions are required. For example, a system similar to that used in Furby's (a children's toy) could be used – a picture of a Furby is shown in Figure 4-2.

Figure 4-2 Furby

Furby's are a type of child's toys that talk. If you have more than one Furby you can get them to talk to each other by "synchronising" them – you simply touch the contacts on the bottom of the toys together, some information is exchanged and they start talking to each other.

It was decided to develop a simple pairing mechanism loosely based on the "Furby" concept in which the information required to pair Bluetooth devices would be exchanged over a serial link when the contacts on the two units are touched together. The use of an Infrared link and an Inductive coupling solution would also be investigated.

The use of a serial link is further supported in a recent Bluetooth SIG Security White Paper [49] in which it was stated "we also recommend that the user be in a "private area", before using the pairing procedure from the Bluetooth Baseband Specification". The White Paper goes on to suggest that "An alternative approach for secure pairing is

to provide a physical serial port interface". The weakness of Bluetooth security with regards to pairing is also highlighted by Jakobssen and Wetzel [3].

Clearly it is imperative that Bluetooth pairing is sufficiently secure to protect the user/owner, is user friendly and that the security measures provided are highly visible to the user to give them peace of mind and confidence in their Bluetooth device.

4.7 PMG – Personal Mobile Gateway Device

During the course of this work it emerged that the GVC Corporation has produced the "IXI Platform" [51] which is based on a device called a PMG (Personal Mobile Gateway). The PMG is similar the PAN Gateway described above. It is interesting to note that the PMG has a minimal user interface with just a power button and an L.E.D. as shown in Figure 4-3.

Figure 4-3 IXI's PMG (Personal Mobile Gateway)

4.8 Chapter Summary

Chapter 4 has discussed and evaluated the aims and requirements of the PAN Gateway together with the possible Man Machine Interfaces that could be used to fulfil the requirements. A new concept for improving the usability of Bluetooth Pairing was suggested and an example of a device that is similar to the PAN gateway has been summarised.

A minimal MMI, consisting of a single button and an L.E.D to indicate whether the device is switched on was found to be the optimal solution provided that a more usable method for pairing Bluetooth devices could be developed; this configuration satisfied the requirements of the PAN Gateway discussed. A new concept for Bluetooth Pairing was proposed based on using a serial link across simple electrical contacts to exchange the data required for pairing. Finally, an existing device similar to the PAN gateway has been introduced.

Following the discovery of the poor usability of the Bluetooth Pairing procedure, the focus of the research changed in order to develop an intuitive Bluetooth pairing method for use in all Bluetooth devices but in particular for the PAN Gateway. The system that was developed is discussed in the remaining Chapters of the thesis.

CHAPTER 5 THE "TOUCH AND FIND" SYSTEM

Chapter 5 develops the concept proposed in Chapter 4 for improving the Usability of Bluetooth pairing by using a serial link into the "Touch and Find" System. In this chapter an overview of the "Touch and Find" system is given together with a development plan and the system requirements. The design of the protocol that specifies the signal flow between the devices is described and the development of the top-level software task, the main PLP task is documented. Finally, the testing of the main PLP task is explained.

5.1 OVERVIEW OF THE "TOUCH AND FIND" SYSTEM

The results of the Usability Study (as discussed in Section 3.6) show that the usability of Bluetooth devices in general needs to be improved considerably in order to meet user expectations and particularly to provide a good "Out of Box" experience. The primary problem in terms of the "Out of Box" experience is the difficult and non-intuitive Bluetooth pairing procedure that must be carried out before any connections are made between two Bluetooth devices. This process is difficult to understand and can be very time consuming.

The concept is to create a system such that by touching together the two devices to be paired, the information that must be exchanged to pair the devices is exchanged over a serial link enabling pairing without needing to go through the usual procedure. By using the new "Touch and Find" system it is hoped that this pairing procedure will become simple, intuitive and even "granny proof"!

5.1.1 Development Plan

The system was developed using "C" in Borland C++ and TTPCom's development system comprised of a "Mad Cow" Evaluation Board (EVB) and Genie[9]. Clearly the interface to the Bluetooth stack would need to be examined and a protocol for the required signal flow to achieve pairing would need to be developed. Initially a (crossed-over) serial cable would be used as the hardware interface between the two

[9] Genie is a test tool which interprets data streams captured from various interfaces of a protocol stack under test, according to filters which the user can set up and presents the results in terms of standard protocol signal primitives. It provides a method for a test engineer to observe the performance of the system under test and compare it with the system specification.

devices; following a successful software implementation, the hardware would be developed.

The development of the "Touch and Find" system was to be carried out as follows: -

1. Investigation into required Bluetooth Interface.
2. Design of basic signal protocol (Pairing Link Protocol).
3. Design and Implementation of Software on PC.
4. Design and Implementation of Hardware.
5. Implement and Test "Touch and Find" system on Evaluation Board.
6. Incorporate "Touch and Find" into a prototype device.

5.1.2 Requirements of "Touch and Find"

The requirements of the "Touch and Find" system are outlined below: -

1. The "Touch and Find" process must be initiated on demand from a higher-level application.
2. It must **not** require a display screen user interface.
3. "Touch and Find" should provide the data to be communicated in a manner compatible with the hardware to be used to transmit the data.
4. It must form a robust serial link between devices.
5. It must return the necessary data to the initiating application.
6. It must provide a high level of security,
7. "Touch and Find" should be a quick process.

5.1.3 System Concepts

The "Touch and Find" system consists of both hardware and software sections. The software must interact with the existing Bluetooth stack to provide the information required to create the link and then output the data in such a way that the hardware can transmit it to the device with which it is to be paired. This is defined in the Pairing Link Protocol (PLP) developed by the author and described in the next section. Initially the hardware will be implemented using a standard RS232 cable. Other methods of communication including simple electrical contacts, inductive coupling and Infrared communication will be investigated later on in the project.

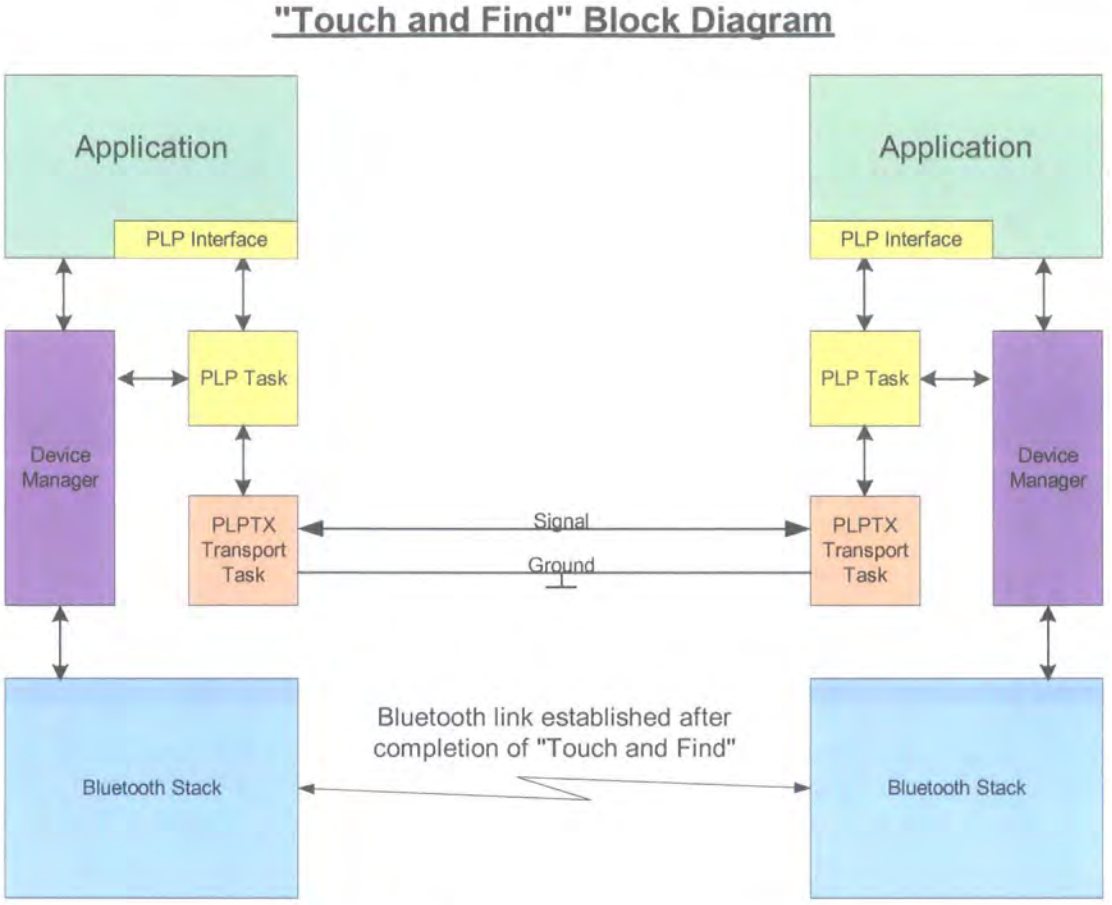


Figure 5-1 Touch and Find Block Diagram

5.2 Main PLP Task

The main PLP task acts as a central hub of communication for the "Touch and Find" process. It initiates all communication by creating and sending signals and by processing incoming signals. A three level design was envisaged as shown in Figure 5-1. The application layer, sitting on top of both the Device Manager and the main PLP task would require a small modification to give it the necessary interface to both initiate the "Touch and Find" process and to use the data it returns.

5.3 Main PLP Task Requirements

The main PLP task must satisfy the following requirements: -

- Adhere to the Pairing Link Protocol (PLP).
- Initiate on request from higher layer.
- Satisfy the interface requirements of lower layers.

- Produce a random number for use as a link key.
- Retrieve Information as required by the PLP protocol from the Bluetooth Device Manager.
- Allow the processor to continue with other tasks, e.g. main Bluetooth operations.
- Complete in a sufficiently short time frame as to be compatible with a handheld link.
- Be robust and resistant to a "bad connection" caused by dirty contacts and corrosion.
- Be independent of the physical method of communicating the data.
- Minimise the amount of data to be transferred across the serial link.

5.4 BLUETOOTH INTERFACE

The processes required to set up a Bluetooth Link were investigated using Wisdom[10] software. It was found that the steps required to create the first link between a pair of devices were significantly more complex than those required to establish the next [separate] link between two particular devices. A transcript of the signals viewed on the log in each of the two situations can be seen in Figure 5-2 and Figure 5-3.

By comparing the transcript of signals sent, shown in Figure 5-2 and Figure 5-3, it is clear that once a connection between two particular devices has been made establishing the second link requires less signals. This is shown by the smaller number of signals present in Figure 5-3 than in Figure 5-2. From the transcript and the Bluetooth Specification [52], it is clear that this is because when the system first tries to establish a link, the upper HCI (Host Controller Interface) layer asks the Device Manager for a link key. However as this is the first connection between these devices no link key exists. With no link key to use the system then uses a PIN key instead. However when the second connection is made, the link key is present and therefore the pin key does not need to be used. This means that the simplest way to create a connection does not use a PIN key, but is instead based on the Link Key. It was concluded that the simplest solution for the PAN Gateway would be to create a new connection by generating and sharing a new link key.

[10] Wisdom is a PC based GUI to allow you to drive a standard Bluetooth device using the standard Bluetooth Host Controller Interface (HCI) over a serial or usb link. provides a User Interface for use with Bluetooth.

Wisdom Log for first link to be established between two devices, no existing Link Key

Master		Slave	
Signal Type	Direction	Signal Type	Direction
HCI Create Connection	DM -> HCU	Connection Request	HCU->DM
Connection Request	HCU->DM	HCI Accept Connection Request	DM->HCU
Link Key Request	HCU->DM	HCI Connection Progress	HCU->DM
Link Key Request Negative Reply	DM -> HCU	HCI Pin Code Request	HCU->DM
HCI Link Key Done	HCU->DM	Pin Code Request	DM-> HPTEST
HCI Pin Code Request	HCU->DM	Pin Code Request Response	HPTEST-> DM
Pin Code Request	DM-> HPTEST	HCI Pin Code Request Reply	DM->HCU
Pin Code Request Response	HPTEST-> DM	HCI Pin Code Done	DM->HCU
HCI Pin Code Request Reply	DM->HCU	Link Key Notification	HCU->DM
HCI Pin Code Done	HCU->DM	Connection Complete	HCU->DM
Link Key Notification	HCU->DM		
Connection Complete	HCU->DM		

-> = signal travels in the direction of the >.

Figure 5-2 Wisdom Log 1

NB: The following settings were used in Wisdom: - Authentication Enabled, Variable Pin (entered by user), Security Level 3.

Wisdom Log for subsequent connection, i.e. with existing Link Key

Master		Slave	
Signal Type	Direction	Signal Type	Direction
HCI Create Connection	DM -> HCU	Connection Request	HCU->DM
Connection Request	HCU->DM	HCI Accept Connection Request	DM->HCU
Link Key Request	HCU->DM	HCI Connection Progress	HCU->DM
Link Key Request Reply	DM -> HCU	Link Key Request	HCU->DM
HCI Link Key Done	HCU->DM	Link Key Request Reply	DM->HCU
Connection Complete	HCU->DM	HCI Link Key Done	HCU->DM
		Connection Complete	HCU->DM

Figure 5-3 Wisdom Log 2

NB: The following settings were used in Wisdom: - Authentication Enabled, Variable Pin (entered by user), Security Level 3.

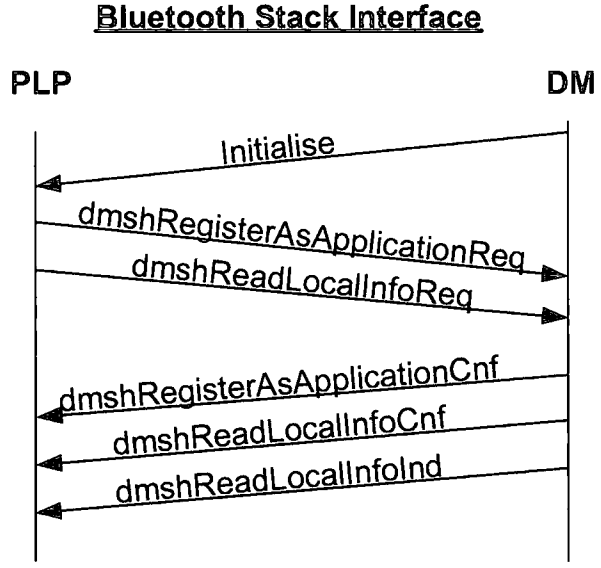


Figure 5-4 Bluetooth Stack Interface Diagram

In order to interface correctly with the existing Bluetooth Stack, the signals shown in Figure 5-4 need to be sent. The process proceeds as follows: -

1. An "Initialise" signal must be sent from the Device Manager to new main PLP task to initiate the process.
2. PLP main task must send dmshRegisterAsApplication to the Device Manager.
3. Send a signal from the PLP main task to the Device Manager to request the local device information (Bluetooth Address and Link Key).
4. Device Manager returns a confirmation of the local information request.
5. Device manager returns with the local information.

5.5 Pairing Link Protocol Concepts

There were two main solutions designed for the main PLP task (for design purposes the signal transport carried out by lower level tasks was assumed). Two concept solutions were devised, to ensure that the different available options were explored and in case no method allowing full duplex communication over a two-contact/wire link could be devised.

The first solution was based on half duplex communication. This solution allowed a Master and a Slave to be determined, which meant that a single link key could be generated and then shared (across the physical link) for use in pairing the Bluetooth devices. The second solution was based on full duplex communication; it worked using a broadcast type system in which both Bluetooth devices created a packet containing the Bluetooth address, friendly name and a randomly generated link key – both devices then transmitted this packet of data and when a device has a copy of both its own randomly generated link key and the other device's, it simply selects the key with the highest numerical value for use in pairing. The two solutions are described below: -

5.5.1 Half Duplex Design

In this design, half duplex communication across the physical layer was assumed. In this solution a Master and Slave are established and the signals must be sent in a pre-defined order. Initially it is assumed that both devices will have a button that is pressed to initiate communication; hopefully these will be removed later. A flow chart of the half duplex solution is shown in Figure 5-5.

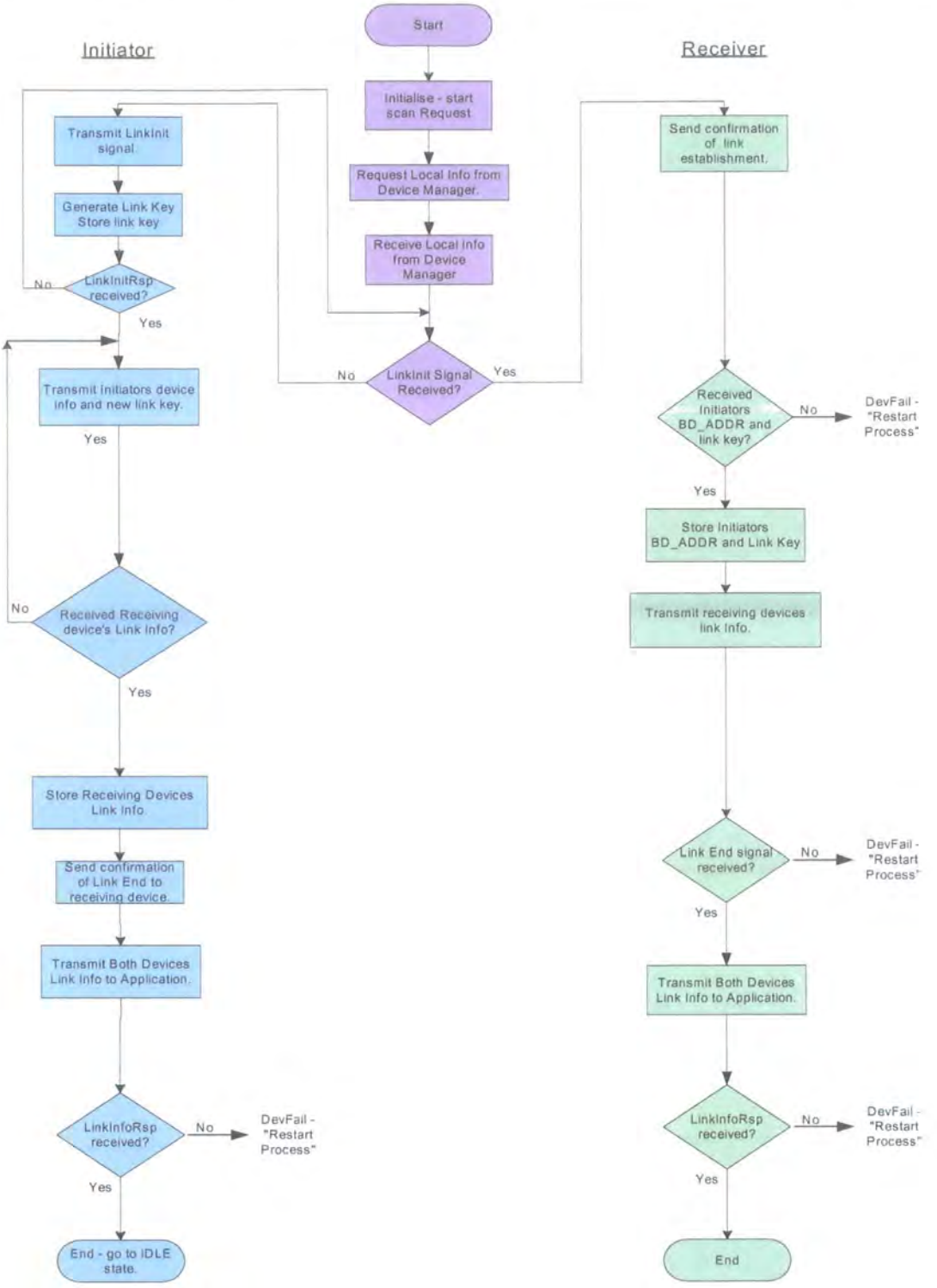


Figure 5-5 Half Duplex Flow Chart

When the button is pressed the device, will initially listen for an incoming signal; after a period of time it will then start to transmit. The device that receives the first signal is defined to be the slave, i.e. the Master initiates communication by being the first to send out its signal. The devices will also transmit a signal to the Bluetooth stack asking for the device Bluetooth Address. When a link has been established and the Bluetooth

Address has been returned, the device initiating the communication (the Master) will generate a random number of the required length for use as a link key. The Master then transmits a packet of data containing both its Bluetooth Address and also the link key. Having received the Master's signal, the receiving device (slave) will then return its own Bluetooth Address, a copy of the link key and the initiating device's Bluetooth Address. The Master confirms that the contents of this data packet is correct and sends a signal to the Slave requesting the end of the link. The Master then sends a signal to the calling application with all the information necessary to pair the devices. Once the slave receives the request to terminate the link, it sends the device information of both devices to its calling application.

5.5.2 *Full Duplex Design*

This design assumes full duplex communication. Once again it is initially assumed that both devices will have a button that when pressed will initiate communication and it is hoped that these will be removed later. The full duplex flow chart is shown in Figure 5-7.

When the button is pressed the device retrieves its own local information, generates a random link key and transmits the link key and device information at regular intervals (controlled by a timer) until it receives a signal from the other system. The full duplex system has four states but does not need to go through all the states; a state timer is used to ensure that the main PLP task does not stay in one of the states for too long. The state diagram for the full duplex solution is shown in Figure 5-6.

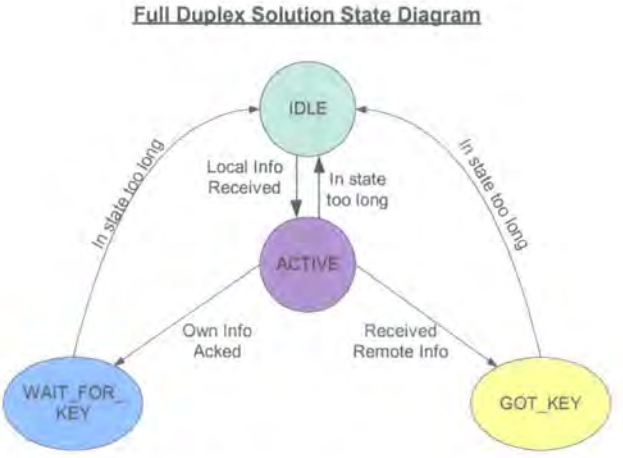


Figure 5-6 Full Duplex State Diagram

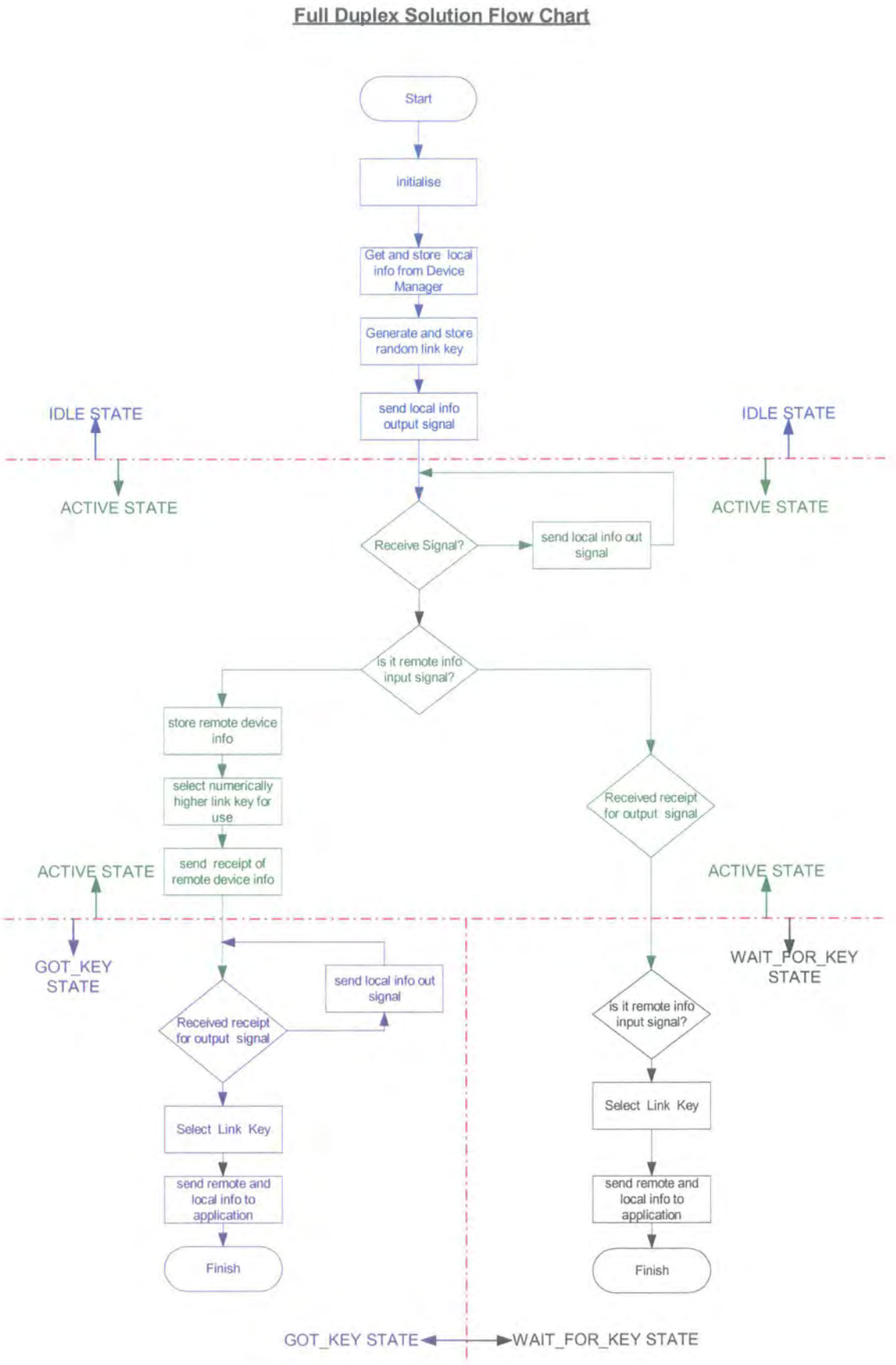


Figure 5-7 Full Duplex Flow Chart

When a device has received the remote device's information and has received a signal that its own information has been received, it has a complete set of information. The complete set of information, containing both local and remote device information, is transmitted as a signal to the application that called the pairing procedure.

5.5.3 Conclusion of Pairing Link Protocol Concepts

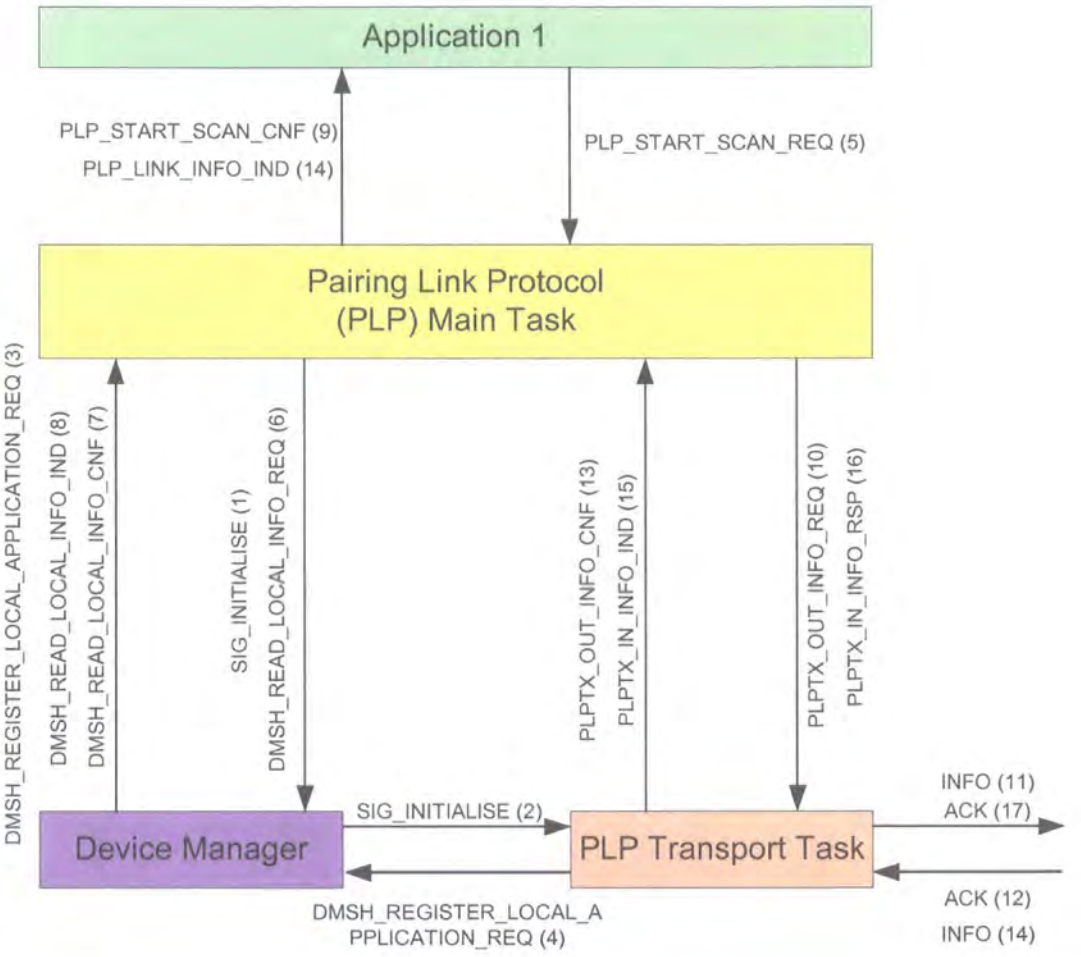
It was decided that the full duplex design would be used for the Pairing Link Protocol, as this was the most robust and would also allow the data to be transferred quickly. The "broadcast" type nature of the full duplex solution created a particularly robust protocol in which many of the signals could be lost and the process would still complete. The "broadcast" type nature also seemed most appropriate for communication across a symmetrical system, i.e. where the two devices that need to communicate are identical and thus can have no permanent Master – Slave hierarchy.

5.6 Design of the Main PLP Task

The main PLP task deals with high level communication and signal flow – it is independent of the physical communication medium used. The main PLP task is the central component of the software for the "Touch and Find" system as shown in Figure 5-8. The diagram shows the order in which signals are sent and which components send/receive as defined in the Pairing Link Protocol. The signal flow defined in the Pairing Link Protocol is shown in Figure 5-9.

Figure 5-9 shows the structure of the "Touch and Find" software and how it interfaces with the Device Manager in the Bluetooth stack. The process can be started by either device 1 or device 2, with the plpStartScanReq signal being sent from the application layer to PLP main task layer. The order of the signals is represented on the diagram by the vertical position of the signals, with the signals at the top of the page being sent first. For simplicity the diagram only shows one of the outgoing information signals; however in reality this signal is sent at regular intervals until the device receives confirmation that it has been received. It should be noted that the number of signals that need to be sent between the two PLPTX (PLP Transport) tasks has been kept to a minimum to make the system as robust as possible.

Full Duplex Signal Diagram



N.B. The numbers to the left of the signal names show the order in which the signals need to be sent, i.e PLP_START_SCAN_REQ (5) is the fifth signal to be sent and is sent from the Application to the PLP Main Task.

Figure 5-8 Full Duplex Signal Diagram

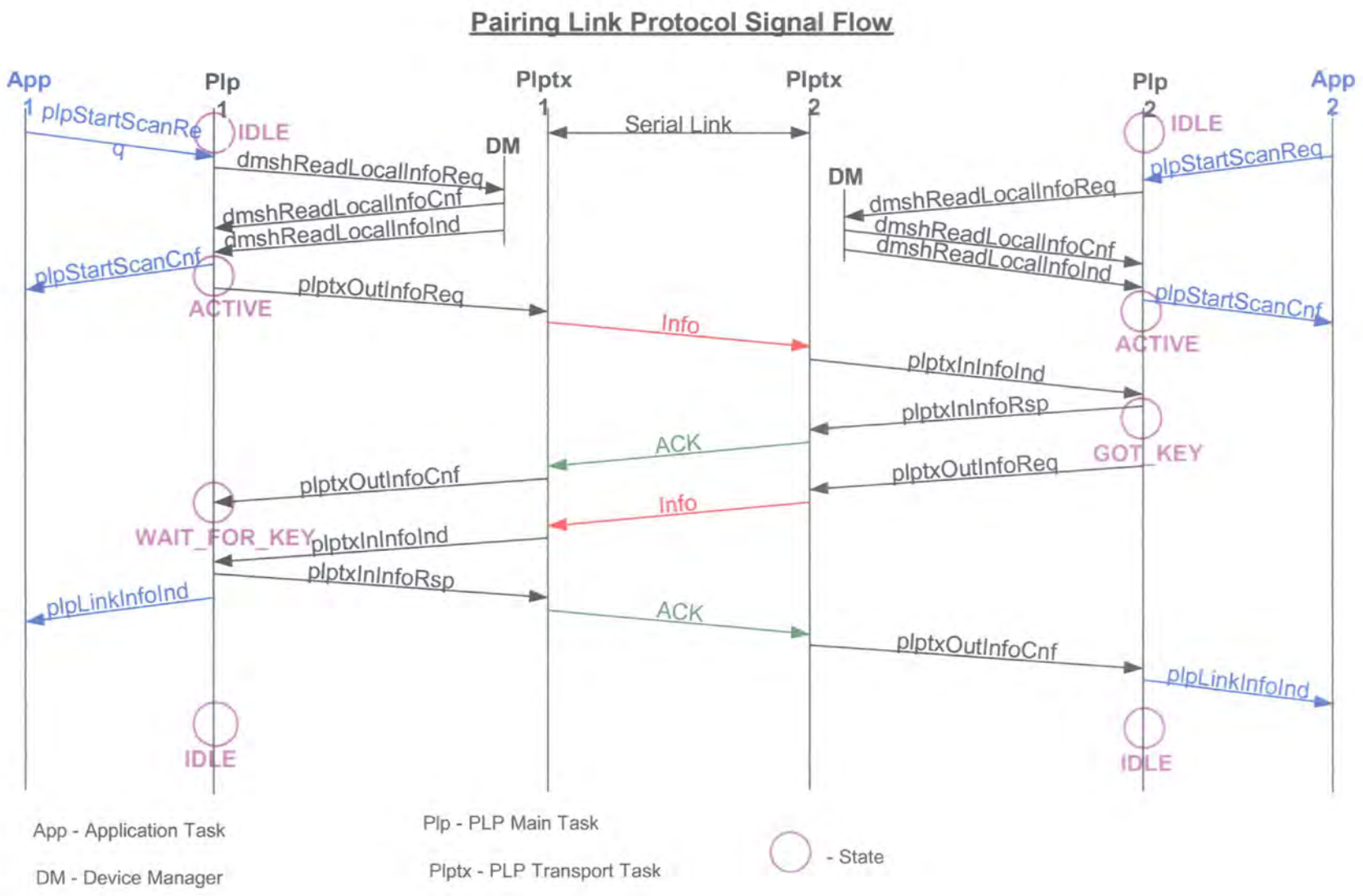


Figure 5-9 Pairing Link Protocol Signal Flow

5.6.1 Main PLP Task Interfaces

The interfaces of the main PLP task shown in Figure 5-8 are described below in Table 5-1.

Interfaces to	Purpose
Application Task	▪ Initiates the "Touch and Find" process. ▪ Receives and uses the link information received.
Device Manager	▪ Sends Initialise signal. ▪ Tasks must register with the Device Manager. ▪ Provides local device information. ▪ Provides Bluetooth Clock for use in generating link key.
PLPTX (Transport) Task	▪ Handles transport mechanism specifics and all low level communication.

Table 5-1 Main PLP task Interfaces

5.6.2 States

The main PLP task was designed to use the states shown in Figure 5-10, Full Duplex Solution State Diagram. This allows the main PLP task to react in a different way to a given signal according to the state it is in.

Full Duplex Solution State Diagram

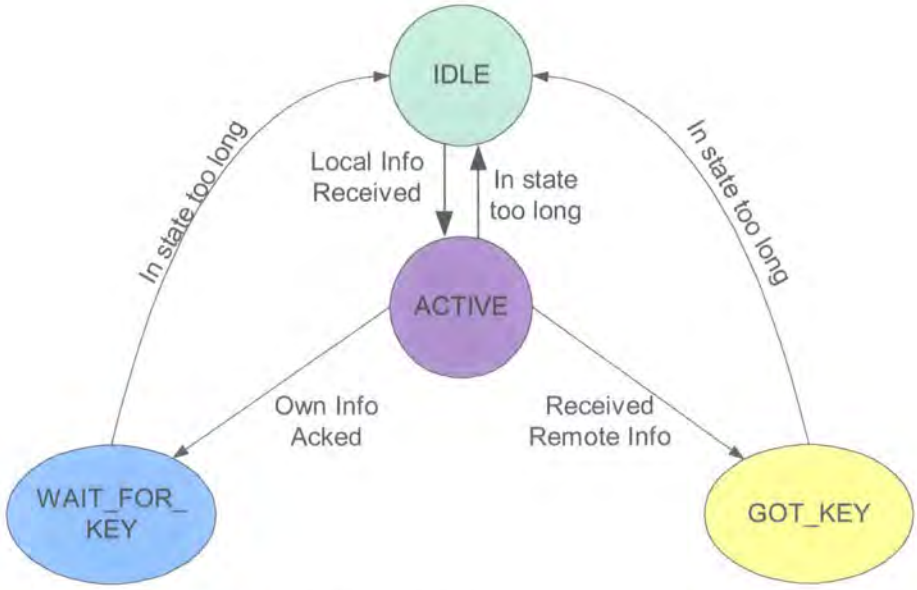


Figure 5-10 Full Duplex Solution State Diagram

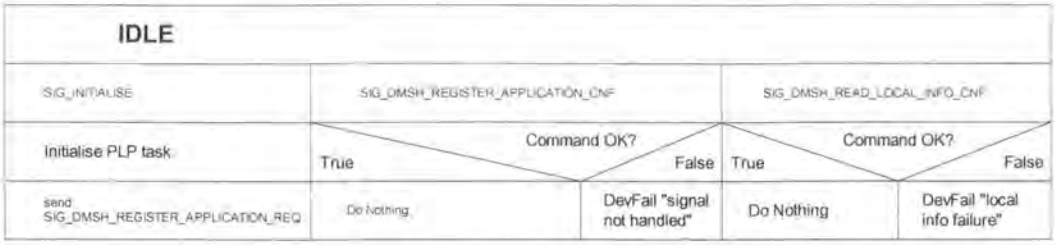
IDLE – this is the base state. The task starts in the IDLE state and returns to the IDLE state once completed. After the "button" to initiate the "Touch and Find" process has been pressed, the local device information has been retrieved and the first signal containing the local information has been transmitted to the PLPTX (transport) task, the ACTIVE state is entered.

ACTIVE – In this state the local device information is transmitted at regular intervals until an incoming signal is received or there is a timeout. If the incoming signal is the remote device information, then the GOT_KEY state is entered (as the main PLP task has "got" a copy of the remote link key). Alternatively, if the incoming signal is a confirmation that the outgoing local information has been received, then the WAIT_FOR_KEY state is entered.

GOT_KEY – In the GOT_KEY state, the local device has received the Remote device's information and now waits to receive confirmation that the remote device has received its outgoing information. The main PLP task continues to output its local information at regular intervals until it receives a confirmation/acknowledgement that the information it has sent out has been received. Once the acknowledgement has been received, the main PLP task selects the link key with the higher numerical value and then sends the complete link information signal to the calling application. The main PLP task then returns to the IDLE state.

WAIT_FOR_KEY – In the WAIT_FOR_KEY state, the local device has received confirmation that its outgoing signal was received by the remote device and now waits for the remote device's information. Upon receipt of the remote device information, it sends an acknowledgement, selects the link key with the higher numerical value and then sends the complete link information signal to the calling application. The main PLP task then returns to the IDLE state.

The basic structure for the main PLP Task was designed using the Nassi-Schneiderman diagrams shown in Figure 5-11, Figure 5-12, Figure 5-13 and Figure 5-14 as these diagrams provided a simple way of representing the different states and case switches.



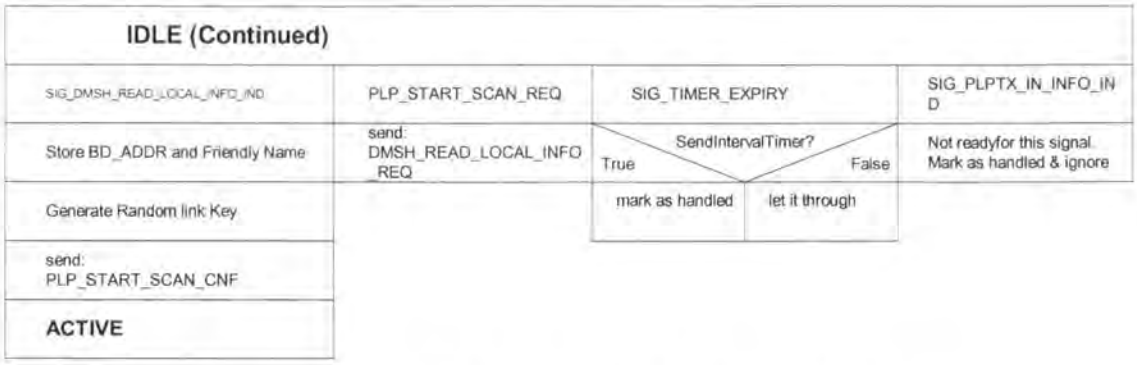
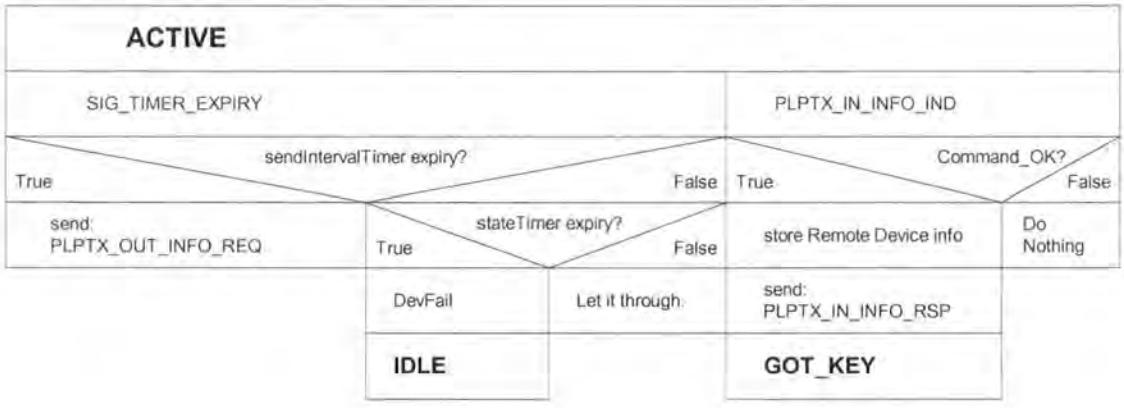


Figure 5-11 Nassi-Schneiderman Diagram of IDLE state.



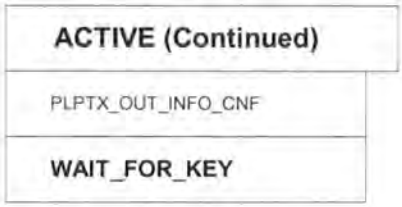


Figure 5-12 Nassi-Schneiderman Diagram of ACTIVE state.

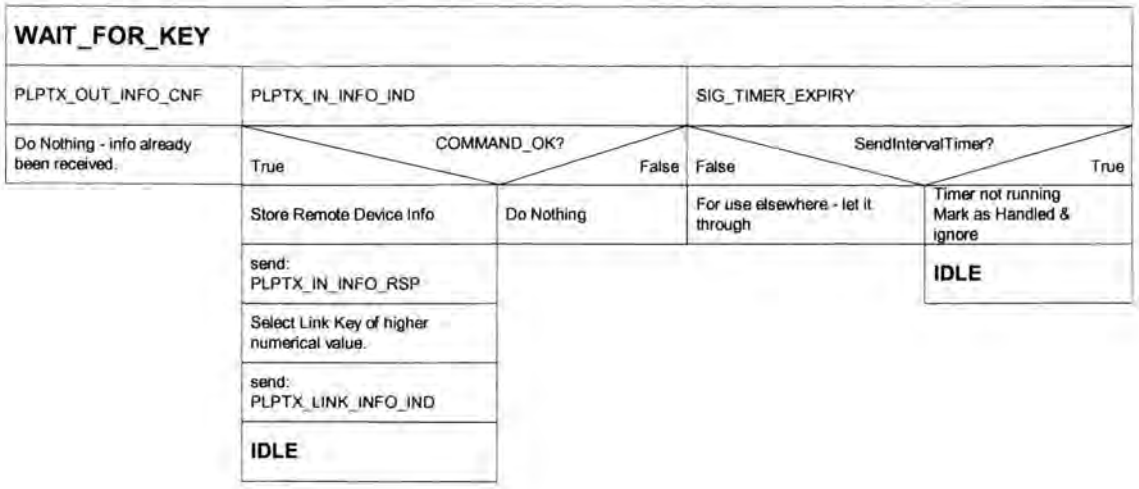


Figure 5-13 Nassi-Schneiderman Diagram of WAIT_FOR_KEY state.

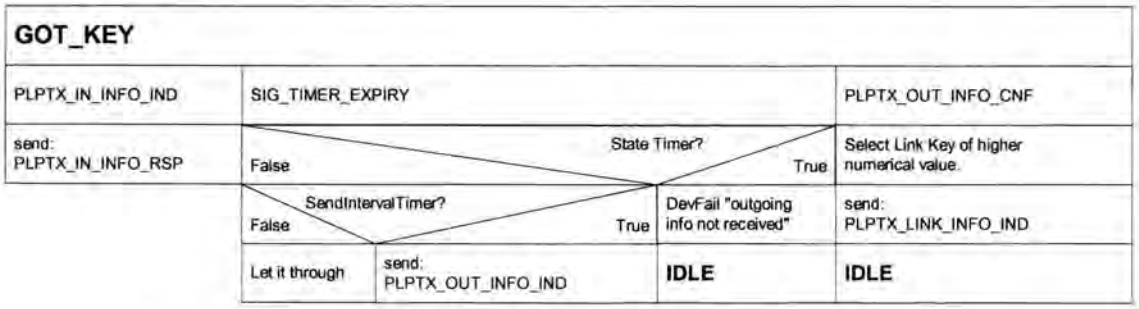


Figure 5-14 Nassi-Schneiderman Diagram of GOT_KEY state.

5.6.3 Important Decisions in the design of the Main PLP Task

Throughout the development of the main PLP task, changes and decisions were made, the most important decisions have been described below: -

IDLE state

- The main PLP task should be initiated following receipt of a SIG_INITIALISE signal from the Device Manager, as this is consistent with the other tasks used in the TTPCom Bluetooth implementation.
- On initialisation the main PLP task should be in the IDLE state.
- Timers should be used to ensure that the main PLP task does not remain in any of the states (other than IDLE) for too long.
- Timers should be used to regularly output the device information as required by the Pairing Link Protocol.
- The process should be initiated from the application calling the "Touch and Find" process. For testing purposes this was carried out from a Test task.

- The main PLP task should remain in the IDLE state until it has sent the first plptxOutInfoReq (outgoing local device information) signal.
- Due to the symmetrical nature of the devices to be connected, both devices send out a link key and then the link key with the highest numerical value is selected by both devices and used to create the link.
- The main PLP task should not request the local information from the Device Manager until it has received the PLP_START_SCAN_REQ (start) signal in order to ensure that the information is up to date – for example the friendlyName might be changed.
- The main PLP task should absorb any PLPTX_IN_INFO_IND (incoming device information) signals received whilst in the IDLE state as a result of the process having just been completed or not been started yet.
- The main PLP task should absorb any SIG_TIMER_EXPIRY's (timer expiry) signals received in the IDLE state that were started by other activities that have now been cancelled.
- Two separate case switches should be used in the main PLP task. The first is to filter out the SIG_TIMER_EXPIRY's (timer expiry's) as a result of the state timer. If the signal received is not a SIG_TIMER_EXPIRY caused by the State timer the second switch was called. The second switch, switches on the State and calls the relevant function for that state.
- The device information should be stored in a single structure, plpLocalDeviceRecord and plpRemoteDeviceRecord for the local and remote devices respectively.

ACTIVE State

- Decided to use variables held in the context to store the remote and local device information, in order to keep all the information together and to make it easier to manage the variables.

WAIT_FOR_KEY State

- The main PLP task should absorb any SIG_TIMER_EXPIRY's received that were started by other processes that have subsequently been cancelled.

GOT_KEY state

- Absorb any extra PLPTX_IN_INFO_IND's (incoming device information) signal that are received.

The use of an "ALL_INFO" state as a final state was considered, in which the signal containing the local and remote device information for sending to the application would be sent. However it was deemed to be unnecessary as there was only one signal to be sent in the ALL_INFO state and no signals were to be received.

5.7 Implementation and Testing of the Main PLP Task

The main PLP task was the first task built. It was built on a single PC with a TTPCom "Mad Cow" Bluetooth Evaluation Board (EVB) connected through the serial port. The EVB was required as it hosts the Bluetooth Device Manager (DM) and the rest of the lower stack.

5.7.1 Isolation Test

The main PLP task was initially tested in total isolation by creating a script to be sent from the TTPCom Genie emulation tool. The script sent all the signals the main PLP task required in order to complete the process in the correct order and had some delays in between the sending of the signals to ensure that the process occurred smoothly, albeit a little slowly.

The script sent the following signals and the code was stepped through using the Borland Debugger.

1. SIG_INITIALISE from Device Manager to main PLP task to initialise the main PLP task.
2. dmshRegisterAsApplicationCnf from Device Manager to main PLP task to register the main PLP task with the Device Manager as required by the Device Manager.
3. plpStartScanReq from the Application to the main PLP task to start the "Touch and Find" process.
4. dmshReadLocalInfoInd from Device Manager to the main PLP task to provide the main PLP task with the local device information that it would normally receive from the Device Manager in response to the request for the local device information.
5. plptxOutInfoCnf from remote device to local main PLP task to simulate the "Acknowledge" signal that would otherwise be sent from the remote device.
6. plptxInInfoInd from remote device to main PLP task to simulate the device information from the remote device being received.

The most significant problem uncovered in the testing procedure was with the two timers used in this task. When a timeout occurs, a SIG_TIMER_EXPIRY signal is sent to the main PLP task Queue. The SIG_TIMER_EXPIRY signal may not be processed until after the timer has been stopped which can lead to the signal being processed in an incorrect state; to avoid this boolean "timerRunning" variables were introduced. Before processing the SIG_TIMER_EXPIRY signal, the "timerRunning" variable is checked and if set to FALSE, the signal is marked as "handled" but not processed.

5.7.2 Test Task Test

In order to ensure that the signals were sent to the main PLP task at the correct time and to create a more realistic test environment, the existing TTPCom Test task code was modified to include some signals for the PLP task. The PLP task was tested with the other TTPCom tasks also running. When Genie is running, it is possible to send signals from the TEST task when a specific button is pressed. In this case the plpStartScanReq signal is sent from the Test task when "6" is pressed and the plptxInInfoInd signal is sent when "7" is pressed.

The Test task was designed to respond to the signals received from the main PLP task in such way as to test all the functions of the main PLP task, i.e. by simulating the presence of another device. The signal flow of the main PLP task interacting with the Test task (after reaching the ACTIVE state) is shown in Figure 5-15.

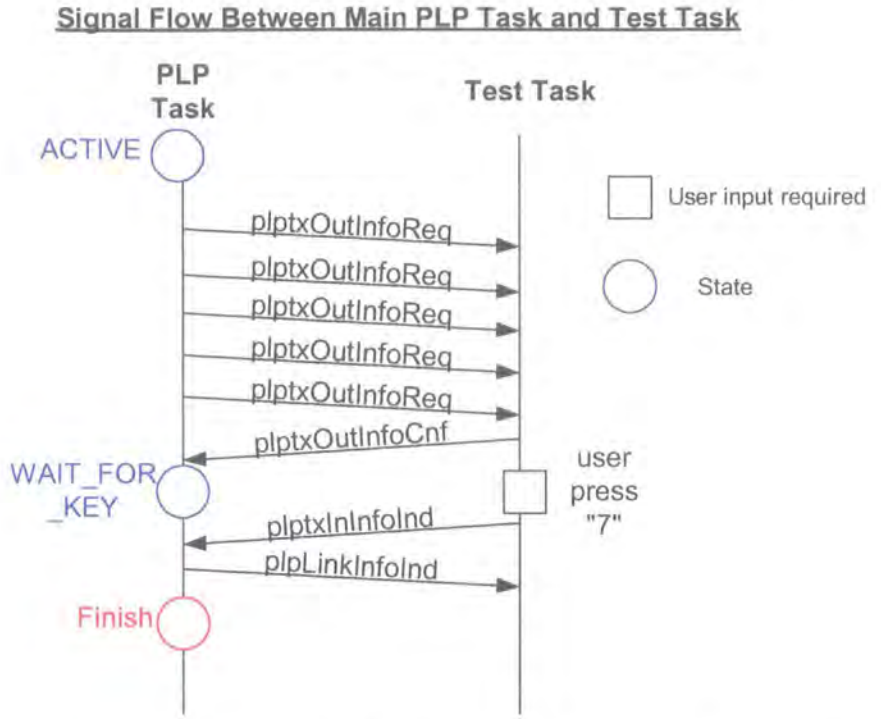


Figure 5-15 Signal Flow between PLP Task and Test Task

When a plptxOutInfoReq (outgoing local device information) signal was received by the Test Task, it ignored the first four copies and on the fifth it generated a PLPTX_OUT_INFO_CNF (confirmation that the outgoing local information has been received) and sent it to the main PLP task. The first four were ignored in order to check that the timer responsible for the interval between the transmissions of the PLPTX_OUT_INFO_REQ (outgoing local device information) signals was functioning correctly.

The main PLP task should now have been in the WAIT_FOR_KEY state, waiting to receive a PLPTX_IN_INFO_IND (incoming remote device information) signal. This signal was sent from the Test task by the user pressing the "7". The main PLP task would then respond to receiving this signal by sending a PLPTX_IN_INFO_RSP (acknowledge) signal and by sending a PLP_LINK_INFO_IND signal to the Test task containing all the relevant device information. The PLP_LINK_INFO_IND signal would normally be sent to the application that called the "Touch and Find" system. If the "7" was not pressed within the interval set in the timeout period of the State timer, the main PLP task would revert back to the IDLE state. However, the test was successful and showed that the main PLP task worked.

5.8 Chapter Summary

In Chapter 5 the "Touch and Find" system and the Pairing Link protocol is introduced. The "Touch and Find" system is a novel method of Bluetooth Pairing using a serial link that is designed to improve the usability of Bluetooth Pairing. A development plan for the "Touch and Find" system was discussed in addition to the requirements of the system. The basic architecture for the "Touch and Find" system was introduced and the required interface with the Bluetooth stack was investigated. The signals required to create a connection were investigated and it was decided that the "Touch and Find" system should create a Bluetooth connection using a Link key rather than a PIN number.

Two different methods of exchanging the necessary information for the "Touch and Find" system based on full duplex and half duplex communication were discussed and it was concluded that the full duplex solution was the best. The full duplex concept was then developed by the author into "The Pairing Link Protocol" which specifies the signal flow between the various sections of the "Touch and Find" system.

Chapter 5 goes on to show the development of the main software task, the PLP task which adheres to the Pairing Link Protocol and interfaces with the existing Bluetooth stack. Finally the successful implementation and testing of the main PLP task is described.

In the following Chapter the PLP Transport task (PLPTX) is described, including the design, implementation and testing of the PLPTX task. The PLP Transport task provides the link between the main PLP task and the hardware that is used to link the two devices using the "Touch and Find" system.

CHAPTER 6 THE PLP TRANSPORT TASK

Chapter 6 introduces the PLP transport task, its requirements and design and shows how the software was developed. The interface to the Windows serial port is described in detail. The methods used to transmit the data and process the received data are described and finally the implementation and testing of the PLP transport task are covered.

The PLP transport task has been named the PLPTX task but it does also include the receive sections of the code. The PLPTX task receives signals from the main PLP task (described in Chapter 5) and converts them into a suitable format for transmission across a standard Windows serial port (see Figure 6-1). The PLPTX task also has to be able to receive signals across the serial link and decode them and create a suitable signal to be sent to the PLP task.

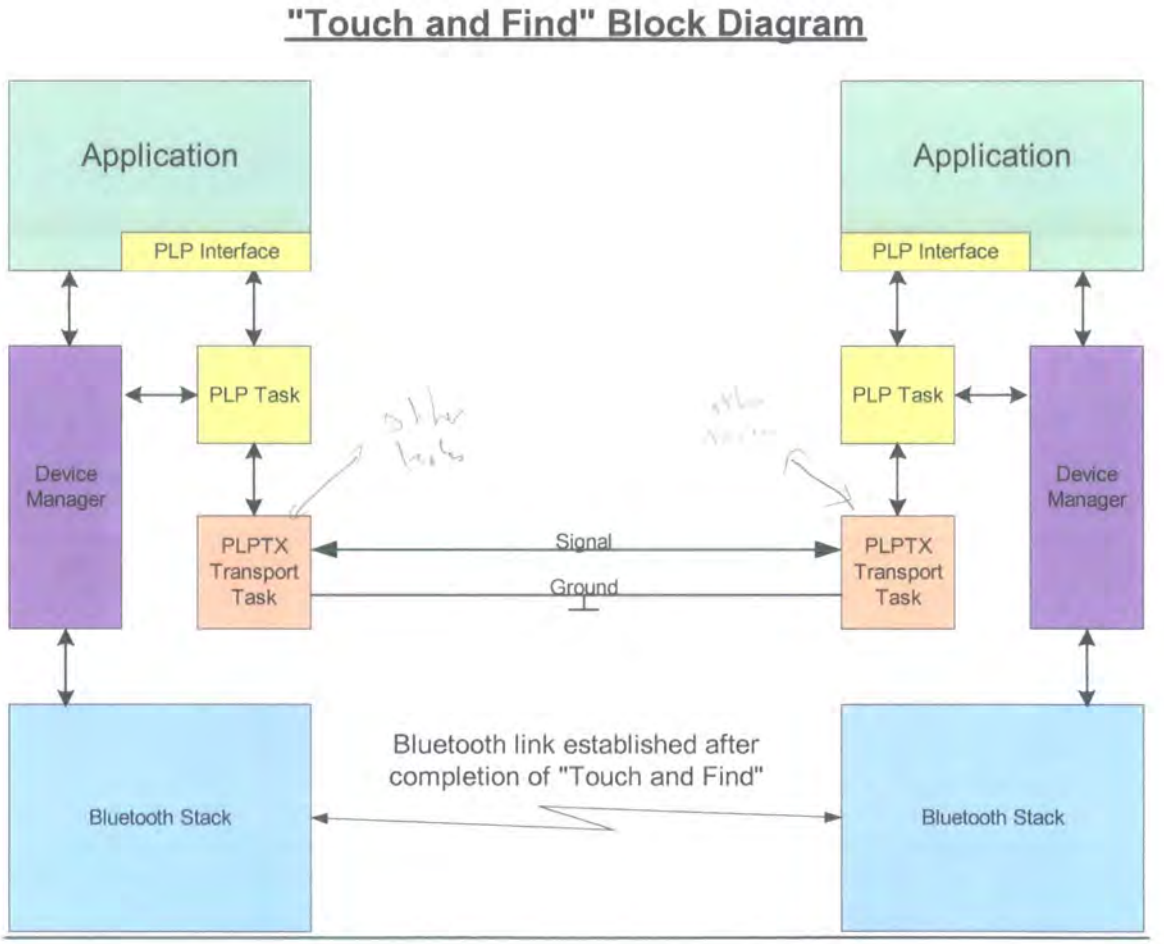


Figure 6-1 "Touch and Find" Block Diagram

6.1 PLP TRANSPORT TASK REQUIREMENTS

The PLP Transport task (PLPTX) task is the module that provides the interface between the main PLP task and the physical layer. The PLPTX task should be independent of the physical medium used. Initially the PLPTX task should be developed to use the Windows serial port interface and must generate signals in a suitable format for transmission and processing at the receiving terminal (i.e. packets should be designed such that they can be simply processed). The PLPTX task should carry out the lower level task associated with serial link maintenance and signal processing.

It is important that the PLPTX (transport) task should not monopolise the CPU time, i.e. it should allow other Bluetooth tasks to have processor time as required; the Windows serial port must be "non-blocking". The PLPTX task must support full duplex communication and must be capable of decoding and processing the received signals and sending signals to other tasks as required.

6.2 PLPTX TASK DESIGN

The PLPTX task has five main areas of functionality; the serial Interface, processing inputs from the other tasks, processing inputs from the windows serial port, generating and sending signals to the windows serial port and generating and sending signals to the other tasks. See the PLPTX Block Diagram shown in Figure 6-2 for details.

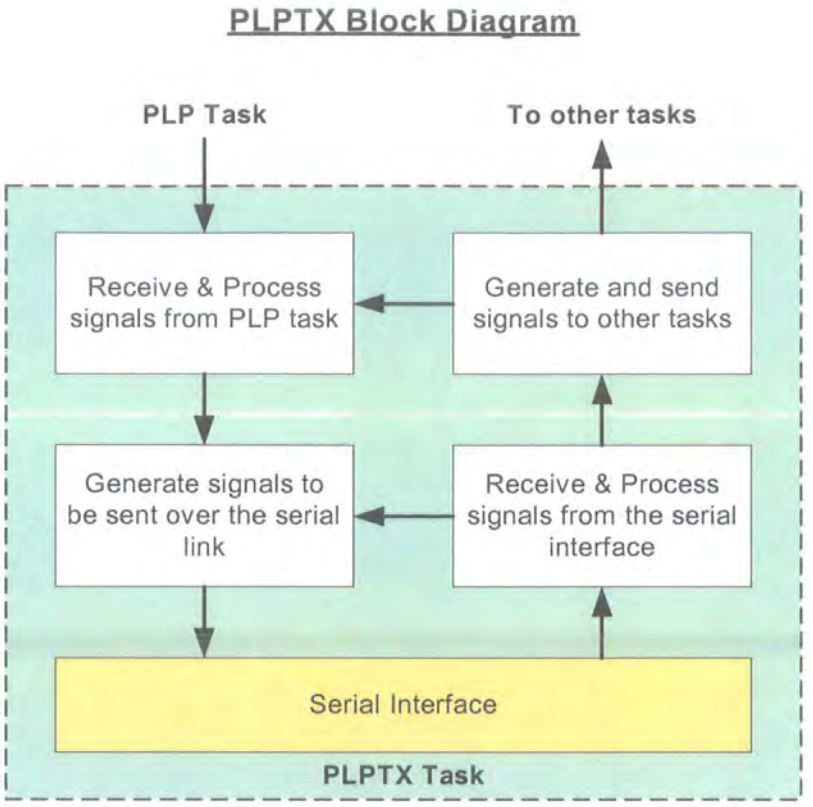


Figure 6-2 PLPTX Block Diagram

6.3 SERIAL INTERFACE

The serial interface is an overlapped Windows serial port. Overlapped input/output (I/O) was used rather than Non Overlapped Input/Output as this (overlapped I/O) is non-blocking, i.e. whilst waiting for a read/write to complete, the CPU is freed up to work on other processes. Overlapped I/O allows reading and writing to be done simultaneously through the use of threads. The code is event driven.

Three threads are used in the PLPTX Bus transport task: -

1. General I/O thread.
2. Transmit (tx) thread.
3. Read (rx) thread.

The general I/O thread deals with the creation and processing of signals that are received or need to be transmitted. The other two threads sit in the transmit/read functions exclusively. It is necessary to ensure that a variable is not changed by the read and write threads simultaneously; this has been achieved by locking the "critical section" as necessary. In this case the "critical section" is the queue that holds the transmit data.

6.3.1 Creating Events

Events are used to signal when data has been received, when the port is ready to transmit more data and when there is more data to be transmitted. "Events" need to be created for the read and write functions (rdEvent and wrEvent) and also to signal to the tx (transmit) thread that there is more data to send (txSignal event). The "CreateEvent" function should be used to create these events.

e.g.

```
plpbuContext.txSignal = CreateEvent ( NULL,/* Security attributes */
                                      TRUE, /* Manual Reset */
                                      FALSE, /* Initial State */
                                      NULL  /* name */  );
```

Handles are used to indicate which file should be read from or written to. Initially these should be set to INVALID_HANDLE_VALUE. Three handles were used, general i/o (input/output), tx (transmit) and rx (receive).

6.3.2 Opening the Port and Setting it up

The procedure for opening up the port and setting it up has been summarised in pseudo code below: -

1. The "CreateFile" function should be used to open the port.
   e.g.

```
ioHandle = CreateFile (PC_COM_PORT,
        GENERIC_READ | GENERIC_WRITE,
        0,              /* share Port */
        NULL,           /* No Security */
        OPEN_EXISTING,      /* How to Create */
        FILE_FLAG_OVERLAPPED,      /* File Attributes - No overlapping */
        NULL        /* Handle of file with attributes to copy */ );
```

2. Get the current Device Control Block (DCB) Settings using "GetCommState".
   e.g. GetCommState (handle, &dcb);

3. Fill in the Device Control Block.

   e.g.

   ```
   dcb.DCBlength = sizeof (dcb);       /* sizeof(DCB)*/
   dcb.BaudRate = 9600;     /* current baud setting - 9600*/
   /* etc....*/
   ```

4. Set the DCB settings using "SetCommState".

   e.g. portReady = SetCommState (handle, &dcb);

   /* if portReady == 1 command was successful */

5. Setup the Input and Output Buffer lengths using "SetupComm".

   e.g. SetupComm(handle, input buffer length, output buffer length);

6. Set timeouts for read and write operations.

   e.g.

   ```
   /* declare */
   COMMTIMEOUTS            timeoutsDefault;

   /* set default timeouts */
   timeoutsDefault.ReadIntervalTimeOut = MAXDWORD;
   /* etc... */
   ```

6.3.3 Creating Threads

As described earlier (in Section 6.3), three threads are used (tx, rx and general i/o). The threads were created using the "CreateThread" function, both the read and write threads need to be created.

```
e.g.  plpbuContext.txHandle = CreateThread ( NULL, /*security attributes */
                              0,   /*Stack size */
                              transmitPacket, /* Tx Thread function*/
                              0,   /*Parameter */
                              0,   /*Create flags */
                              &threadId /*Thread identifier*/
                              );
```

6.3.4 Tx Thread Function

In the tx thread the transmitPacket function is executed continually; it was declared as follows:-

```
DWORD WINAPI transmitPacket (LPVOID ptr);
```

In the transmitPacket function the tx thread waits for the txSignal event to be set by calling WaitForSingleObject as follows: -

```
WaitForSingleObject (plpbuContext.txSignal, INFINITE); /* timeout is infinite */
```

Once the txSignal has been set (i.e. there is data to send) the transmitPacket function calls WriterGeneric which does the WriteFile call to write the data to the serial port. WriterGeneric is a fairly standard function documented in MSDN help files. A flowchart of the WriterGeneric function is shown in Figure 6-3.

The WriterGeneric function contains some error trapping for the WriteFile function. In Overlapped I/O the writefile often does not return immediately – this causes WriteFile to return "operation not successful", GetLastError is then called.

If GetLastError returns anything other than "ERROR_IO_PENDING" there is a fault and the process will fail. If it returns "ERROR_IO_PENDING" then the WriteFile was delayed due to the CPU being busy. The WaitForSingleObject function is then called, it waits for the processor to have time to complete the WriteFile, it also allows time for the expected data to be received into the buffer. Once the wrEvent is set, GetOverlappedresult is called to determine if the WriteFile was successful.

6.3.5 Rx Thread Function

The Rx thread function works in a very similar way, it calls ReadGeneric (see Figure 6-3 for ReadGeneric flowchart) passing in the number of bytes to be read. If the data is already available, the ReadFile in ReadGeneric returns immediately. If all of the data is not there, WaitForSingleObject is called, returning either after the timeout interval or when the data has been received. Finally, GetLastError is called to verify that the read operation was successful.

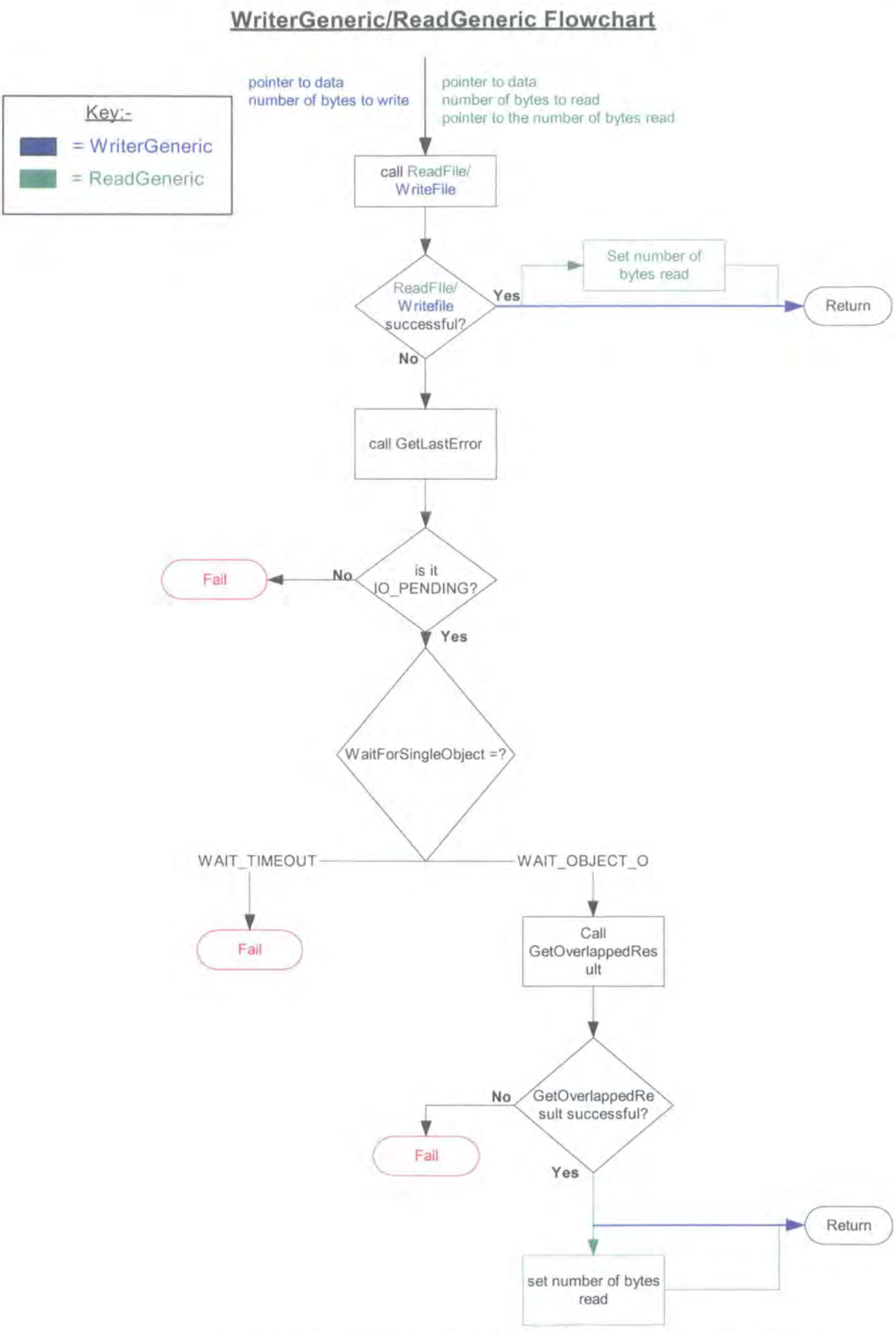


Figure 6-3 ReadGeneric and WriterGeneric FlowChart

6.4 PLP TRANSPORT TASK – WRITING A SIGNAL TO THE SERIAL PORT

The bus task has been designed such that to output any signal it is necessary to create a SIG_PLPTX_BUS_WRITE_DATA_REQ signal, fill it with the necessary data and call the plptxBusWriteData function passing in the newly created signal. This method was used as it provides a standard way of outputting data via the serial port, thus simplifying the process and allowing the serial interface section of the code to be re-used in other modules. From the development point of view, the use of a single signal to output data to the serial port makes it very easy to see when data is being output to the serial port during the debugging process.

The PlptxBusWriteData function then adds the contents (i.e. signal to be sent) of the SIG_PLPTX_BUS_WRITE_DATA_REQ signal to the queue of signals to be output and signals that there is more data to transmit by setting the txSignal event as shown in Figure 6-4.

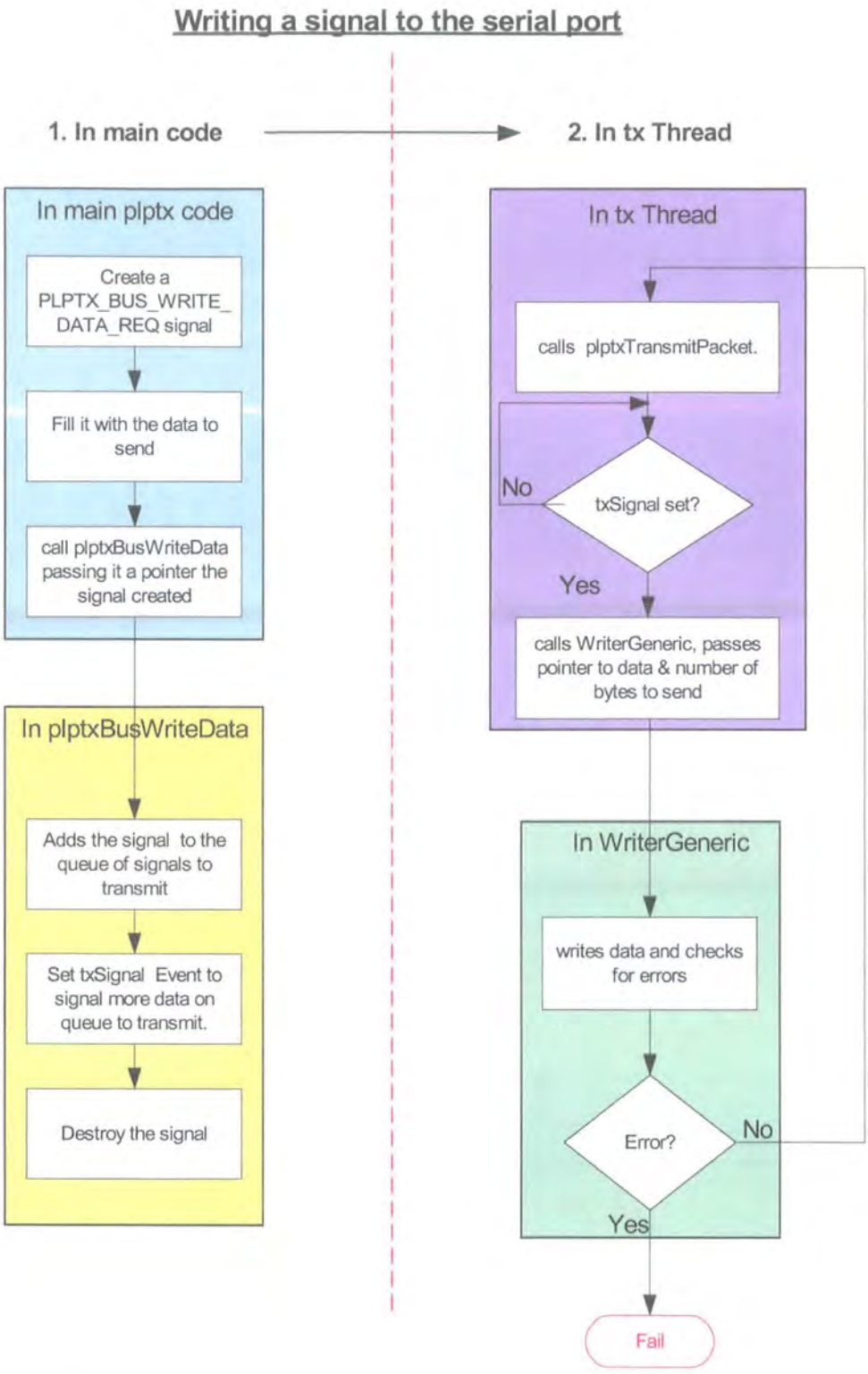


Figure 6-4 Flowchart of writing a signal to the serial port

The steps carried out in each of the functions in order to write a signal to the serial port are summarised below: -

In plptxmn (main plptx code)...

1. A SIG_PLPTX_BUS_WRITE_DATA_REQ signal is created.
2. The SIG_PLPTX_BUS_WRITE_DATA_REQ signal is filled with data/signal to be sent including any header signals required for processing. The PUT_INT family of (TTPCom) functions should be used to put the data into the signal to be sent and the GET_INT family of functions used to read data from received signals as this avoids the problems caused by some systems being little endian whilst others are big endian.
3. The plptxBusWriteData function is called and a pointer to the SIG_PLPTX_BUS_WRITE_DATA_REQ signal that has just been created and filled is passed into it.

In plptxBusWriteData function...

4. The queue of signals to be transmitted is locked as this is a "Critical section".
5. The signal to be transmitted is added to the queue of signals.
6. The queue is unlocked (this is no longer in the "critical section").
7. The txSignal event is set to signal that there is more data to transmit in the tx thread.
8. The signal is destroyed (there's still a copy of the signal on the queue of signals to be sent).

In the tx (transmit) thread (in transmitPacket)...

9. The tx thread waits for the txSignal to be set.
10. WriterGeneric is called and a pointer to the data and the number of bytes to be sent is passed in.

In the tx thread (in writerGeneric)...

11. WriteFile is called (this is the function that writes the data to the serial port.

6.5 PLP TRANSPORT TASK – SIGNAL FORMAT

The signal format was designed to make processing the signal in the receive thread as simple as possible. The signals to be sent over the serial link were designed to start with a 1-byte field containing the packet type. The packet type can be either INFO_TYPE (a signal containing the device information) or ACK_TYPE (an "acknowledge" or "not_acknowledge" signal).

The second field consists of 1 byte that contains the signal type (e.g. PLPTX_BUS_OUT_INFO or PLPTX_BUS_ACK). The rest of the signal then follows. In the PLPTX_BUS_ACK signal the third byte contains the signal that is being acknowledged.

The number of bytes to be received is initially setup to be 1, i.e. the packet type byte. Having received the packet type byte, the size of the signal is known and thus the number of bytes to be read in ReadGeneric can be set accordingly. The bus signal structures used are shown in Figure 6-5.

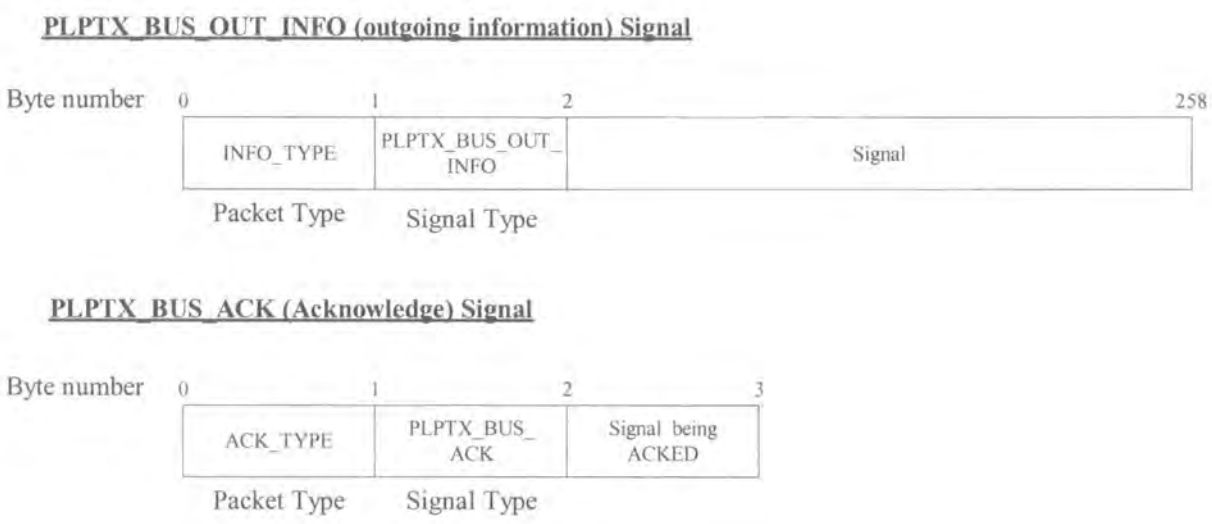


Figure 6-5 Bus Signal Structures.

6.6 PLPTX CODE – READING A SIGNAL FROM THE SERIAL PORT

A flowchart of reading a signal from the serial port is given in Figure 6-7. The receivePacket function is in the receive thread and consists simply of a loop that calls the following: -

1. Call ReadGeneric with the number of bytes to be read – in ReadGeneric, the system waits for the required number of bytes to be received, read them and then returns.
2. Call the plpbuProcessRxData function – this passes a pointer to the received signal into the plpbuProcessRxData function which processes the signal.

As previously mentioned, the number of bytes to be read is initialised to be 1 byte. The receive state rxState is initialised to PLPTX_BUS_RX_PACKET_TYPE (see Figure 6-6 rxState State Diagram). When a signal is received the packet type is determined from the first byte and from this the number of bytes to be read is set. A flowchart of receiving and processing signals is shown in Figure 6-7.

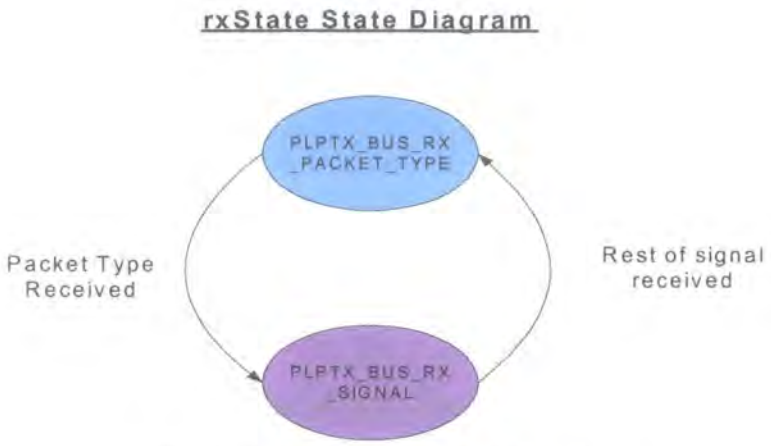


Figure 6-6 rxState State Diagram

Received data is processed in plpbuProcessRxData which is called from the receive Packet function. The plpbuProcessRxData function processes the first byte and determines what type of packet has been received, i.e. whether it is an INFO (information) type packet or an ACK (acknowledge/not acknowledge) type. Depending on what type of packet has been received the number of bytes that are left of the signal to be read are set and the rxState is set to PLPTX_BUS_RX_SIGNAL. The plpbuProcessRxData then returns to its calling function receivePacket.

The ReceivePacket function calls ReadGeneric with the new number of bytes to be read; when it returns plpbuProcessRxData is called. PlpbuProcessRxData initially processes the first byte to evaluate what type of signal it is and then calls the appropriate function to deal with the contents of the signal. The number of bytes to be read is then reset to 1 (for determining the TYPE of packet) and then resets the rxState to PLPTX_BUS_RX_PACKET_TYPE. The receive thread waits in WaitForSingleObject in ReadGeneric for the next data to arrive.

This method of processing the signal has been used as it is simple and can be easily extended if it becomes necessary to add in more packet types or signal types. It relies on the receiving system knowing how many bytes there are in each type of signal but this simply requires a database of the different signal types.

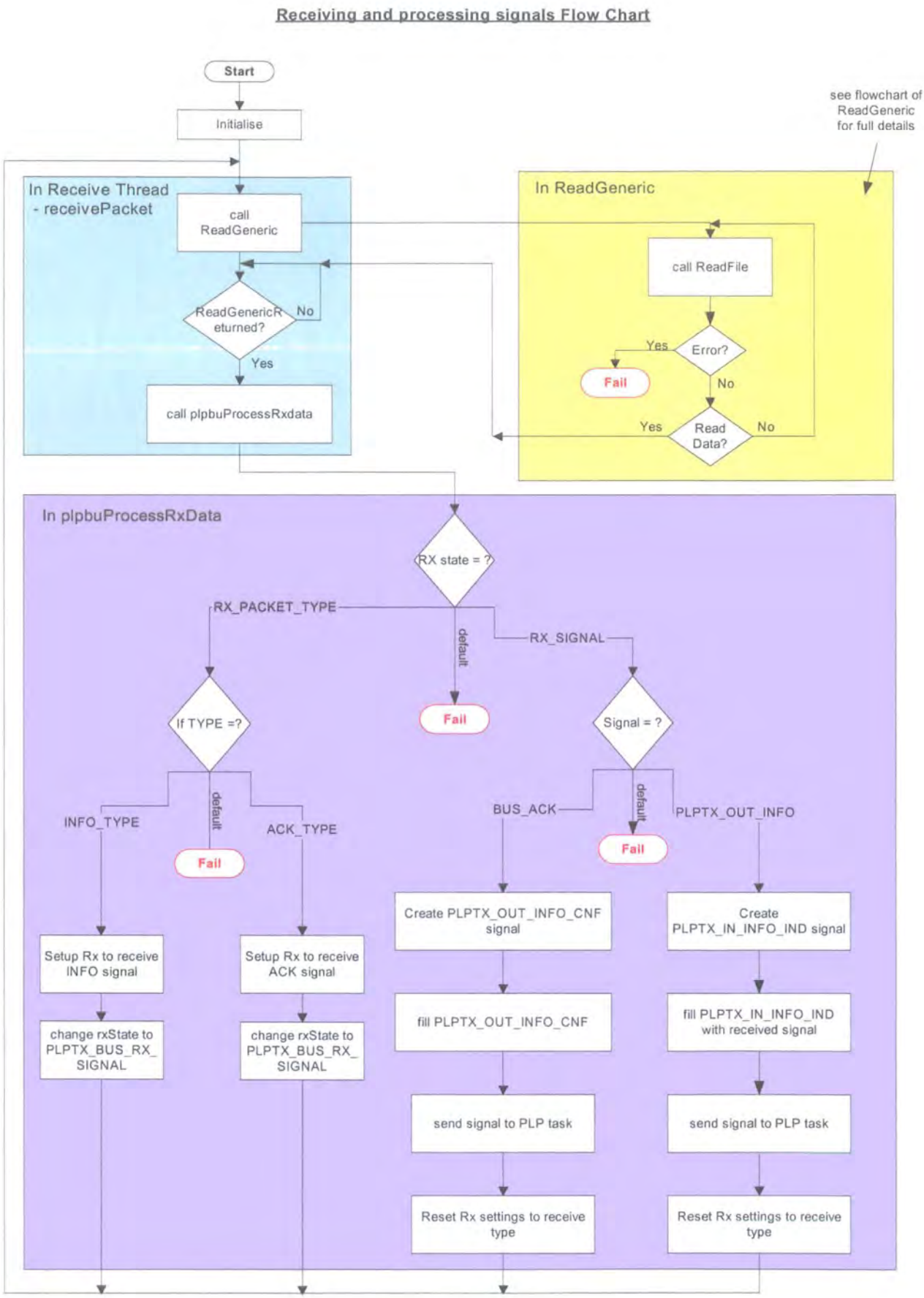


Figure 6-7 Flowchart of Receiving and Processing Signals

Nassi-Schneiderman Diagrams representing the plpbuProcessRxData function are shown in Figure 6-8. The diagrams show that there are three levels of case switches within the plpbuProcessRxData that are used to process the received data.

Nassi-Schneiderman diagrams of plpbuProcessRxData

PLPBU_PROCESS_RX_DATA		
case PLPTX_BUS_RX_PACKET_TYPE	case PLPTX_BUS_RX_SIGNAL	default:
PLPTX_BUS_RX_PACKET_TYPE	PLPTX_BUS_RX_SIGNAL	DevFail (" incorrect type")

PLPTX_BUS_RX_PACKET_TYPE		
INFO_TYPE	ACK _TYPE	default:
set number of bytes to be read	set number of bytes to be read	DevFail (" incorrect type")
reset receive Buffer pointer	reset receive Buffer pointer	

PLPTX_BUS_RX_SIGNAL		
PLPTX_BUS_OUT_INFO	PLPTX_BUS_ACK	default:
create PLPTX_IN_INFO_IND	create PLPTX_OUT_INFO_CNF	DevFail (" incorrect signal")
fill and send to PLP task PLPTX_IN_INFO_IND	fill and send to PLP task PLPTX_OUT_INFO_CNF	

Figure 6-8 Nassi-Schneiderman diagrams of plpbuProcessRxData

6.7 Important Decisions in the design of the PLP Transport (PLPTX) Task.

The most important decisions in the design of the PLP Transport task were deciding which method to use to transmit the data and how the signal structure needed to be designed to facilitate simple processing of the received signal. To send signals to the Windows serial port it was decided to always use a single signal type (a plptxWriteDataReq signal) and fill it with the signal to be sent. This made it very simple to send any type of signal and can easily be extended if other signals are added.

To simplify the processing of the received signals, the packet type byte was used as the first byte in the signal. This allowed the number of bytes in the signal to be quickly determined enabling the rest of the signal to be read. The second byte gives the signal type which determines how the data in the rest of the data in the signal is processed.

This structure was used as it is simple and can be easily extended if other signals are added at a later stage. The received data is processed in the PLPTX task and signals are sent to the main PLP task and to the PLPTX task (i.e. an internal signal is sent) as required. The main PLP task only deals with higher level processing.

6.8 Implementation and Testing

Initially a very basic procedure was used to test the Windows Serial Port. The basic code for accessing the serial port and writing to it was implemented and then tested by connecting the Serial Port via a serial cable to a second PC running Windmill's ComDebug[11]. ComDebug simply acted as a terminal that showed what data was being output to the serial port from the code being tested.

Initially just an "A" was output from the PLPTX (PLP Transport Task) code to see if it was displayed on the ComDebug terminal. When this test was successful, several bytes were output successfully. Having established that the transmit mechanism worked the read mechanism was tested.

The number of bytes to be read was set to 1 and an "A" was output from the ComDebug software running on the other PC whilst the PLPTX code was run in "debug" or "step through" mode looking at the receive thread. The PLPTX task satisfactorily read the "A" and output it on a trace signal in Genie that was used for debugging.

Debugging the code proved to be both difficult and time consuming due to the multi-threaded processes. Borland C++ supports multi-thread debugging; however only one thread can be active at any one time and therefore it is imperative that the correct thread is being looked at during the debugging process. It is important to remove breakpoints from sections of code that may be part of a different thread, otherwise it may be impossible to enter the thread that needs to be stepped through.

Having established that the basic communications mechanisms worked, the mechanism of writing a real signal to the serial port was tested (i.e. using the plptxWriteDataReq signal to output all data) using the ComDebug software. Similarly, the basic structure of the reading and processing of received data from the serial port was tested using the ComDebug software.

[11] ComDebug is a freeware communications debugging tool from "Windmill" that acts as a terminal.

The read/process mechanism was checked using a simple test signal that consisted of the Type byte, the signal name byte and a Bluetooth address as shown in Figure 6-9. The INFO_TYPE packet type was used and the PLPTX_BUS_OUT_INFO signal name was used to verify that the processing function handled the signal correctly.

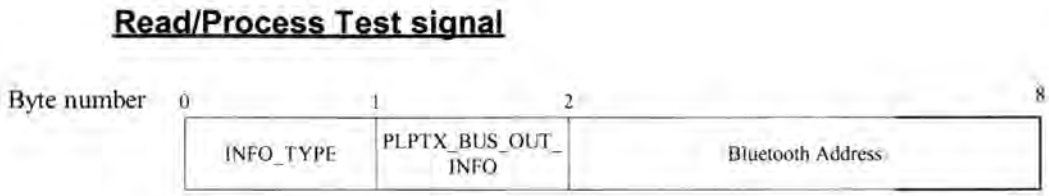


Figure 6-9 Read/Process Test Signal

The rest of the PLPTX task was then implemented based on the architecture described in Figures 5-9, 6-7 and 6-8. The setup used to test the PLPTX task is shown in Figure 6-10.

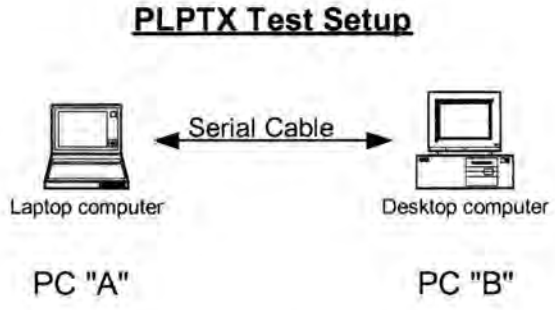


Figure 6-10 PLPTX Test Setup

Initially the PLPTX task was run in isolation on the laptop (PC "A"). Once again the signals required from the device manager were implemented using a script in Genie. The purpose of this test was to ensure that the correct data was being output in the PLPTX_OUT_INFO_CNF signal and to check that if a PLPTX_BUS_ACK (an acknowledge signal for the PLPTX_OUT_INFO_REQ signal) was sent from the comDebug terminal that it was processed appropriately. After a few changes due to having incorrectly calculated the length of the various fields whilst filling the outgoing signal, the results of this test were satisfactory.

The main PLP task was added (PLPTX and main PLP tasks were run on the laptop at this stage) to check that the PLPTX task interacted with it correctly. It was not possible to have all of the other tasks running simultaneously, as this would require having an

EVB (Evaluation Board) connected to each PC using a serial port, which could not be done as the laptop's only serial port was being used for testing the serial link.

The next stage was to run the PLPTX and main PLP code on both PC's to determine if the whole system worked. Once again the Device Manager was being simulated using the Genie script. Both the main PLP task and the PLPTX task were run in "debug" mode and the code was stepped through to verify that it functioned correctly.

Having established that there were no problems with the logic, both PC's were taken out of debug mode and were run normally with both the main PLP and PLPTX tasks running. This test showed that although the "Touch and Find" process was being completed, the tasks did not stop on completion and signals continued to be received. Various solutions to this problem were considered, such as: -

- Clearing the queue of signals to be transmitted.

After consideration it was concluded that this solution would not work as clearing the queue of signals to be transmitted would stop the other device being able to complete the "Touch and Find" process.

- Clearing the receive buffer.

Although clearing the receive buffer may help, it would also prevent the device responding to requests for information from the other device which may not have completed the "Touch and Find" process yet.

- Adding a signal to be sent between the devices to indicate that that device has finished. Only if a device has both sent the plptxOutFinishReq signal and received a plptxInFinishInd will it be able to change back to the IDLE state.

This solution (adding a "finish" signal) was implemented and works well as it ensures that if one device finishes and the other has not, the first device does not go back to the IDLE state but remains in the same state until either it receives a request to finish the link or alternatively the state timer times out. This ensures that it can correctly respond to requests for information from the device that has not yet completed the "Touch and Find" process. There were still some problems with signals being processed in the correct state, but these signals were simply handled and absorbed in the relevant states.

6.8.1 *Addition of Autostart feature*

It was realised that instead of needing to activate the "Touch and Find" process manually, e.g. using a button, it would be beneficial if the "Touch and Find" process started automatically. This "autostart" required the system to be able to detect when it was in the "Connected" state.

The simplest solution was to use some handshaking; it was decided to send a start sequence signal at the beginning of the link and to add two additional states to the plptx task in the form of tx States, the "DISCONNECTED" state for when there was no active serial link and the "CONNECTED" state when there was an active serial link (see Figure 6-11). When in the "DISCONNECTED" state only the start sequences can be transmitted.

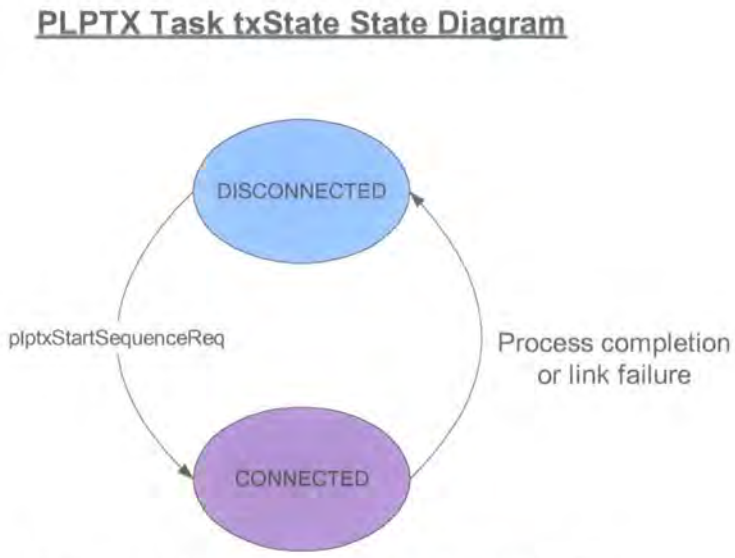


Figure 6-11 PLPTX Task txState Diagram

The start sequence was sent out at regular intervals (using a timer) from both devices' serial ports. When the start sequence (plptxStartSequenceInd) was received, an acknowledge signal would be sent. When a device has received both the plptxStartSequenceReq and plptxStartSequence2Req, a plptxStartSequenceCnf signal would be sent internally to the main PLP task and the plptx state would change to the "CONNECTED" state; the serial link is now active.

The signal flow chart with both the start sequence and the finish sequence shown is shown in Figure 6-12, internal signals are not shown.

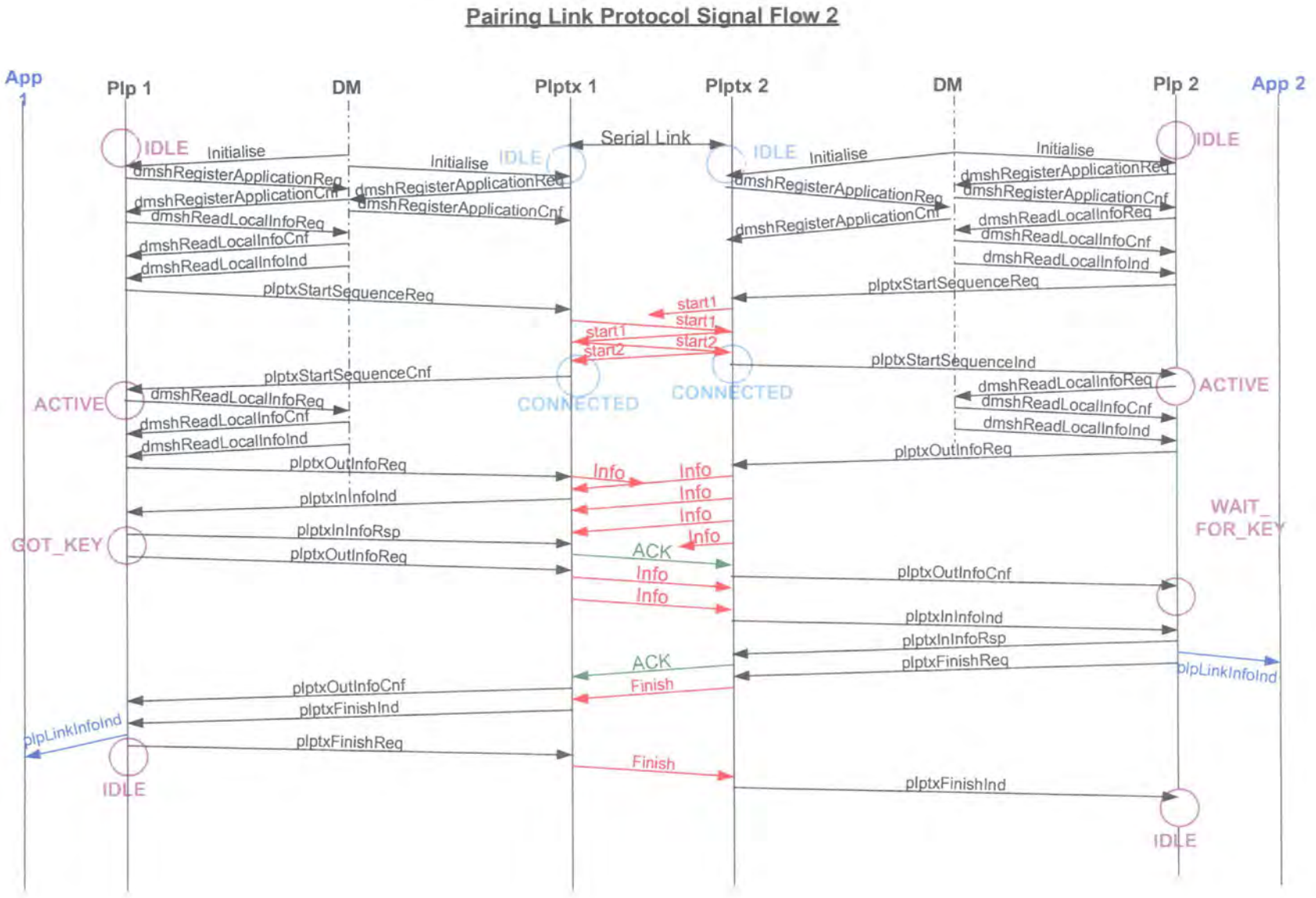


Figure 6-12 Pairing Link Protocol Signal Flow 2

The start sequence was sent at regular intervals from the PLPTX task after the Initialise signal had been received from the Device Manager. The signal flow for the start sequence including the internal signals is shown in Figure 6-13; for simplicity the dmshRegisterAsApplicationCnf signal is not shown and the signal flow in just one device is shown.

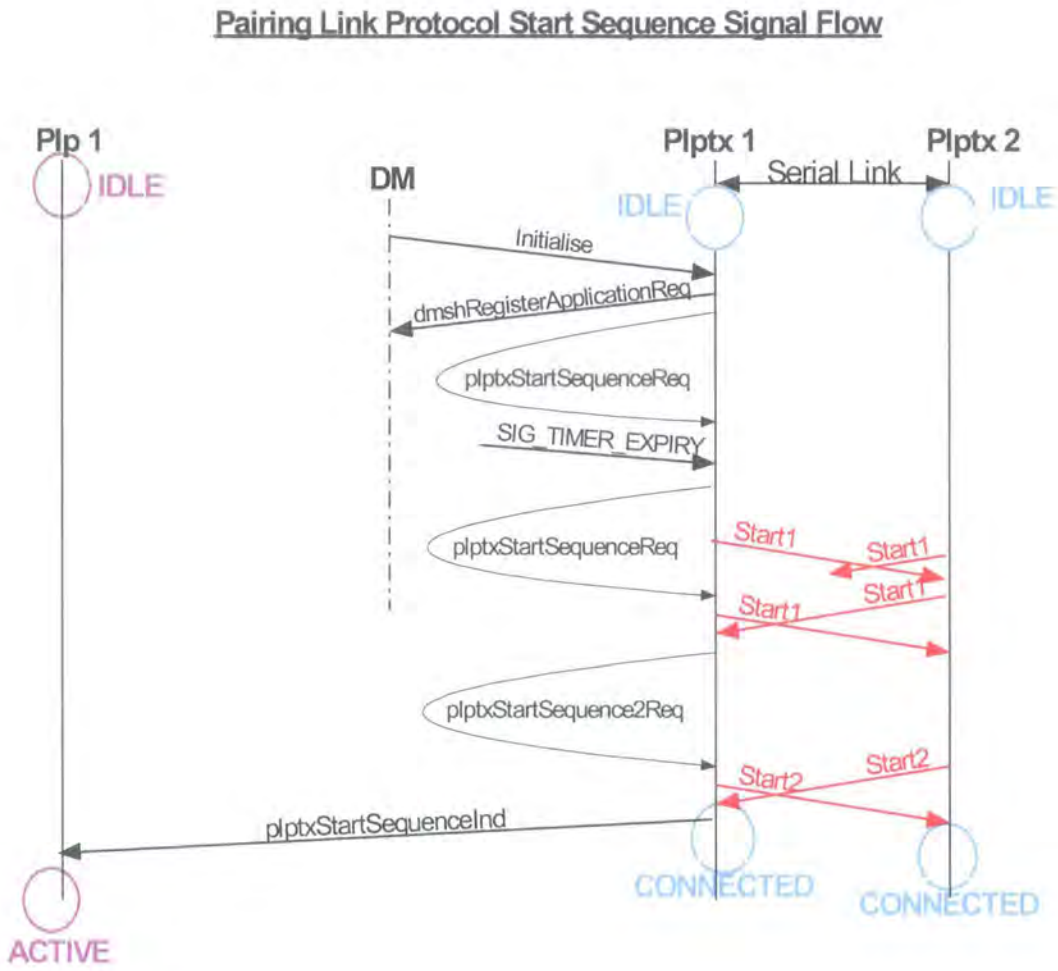


Figure 6-13 Pairing Link Protocol Start Sequence Signal Flow

Having received the SIG_INITIALISE signal from the Device Manager, the PLPTX task creates and sends the plptxStartSequenceReq signal to the PLPTX task (itself) and starts a timer with a timeout period of half a second. When the timer expires the plptxStartSequenceReq signal is created and sent to the PLPTX task. This mechanism of the PLPTX task generating and sending signals to itself, to then be transmitted via the serial port was chosen as it isolates the main PLP task from the low level task of determining if the serial link is connected.

When the PLPTX task receives the plptxStartSequenceReq, it creates and sends a "Start1" signal to the serial port transmit queue that consists of "AAA". When the PLPTX bus receives "AAA", it sends a plptxStartSequence2Req to PLPTX. This causes the PLPTX task to create a "Start2" signal "ABB" and send it to the serial port transmit queue. When "ABB" is received the plptxState changes to "CONNECTED" and a plptxStartSequenceInd signal is sent from the PLPTX task to the PLP task.

By using start sequence packets of "AAA" and "ABB" the packets can easily be processed by the existing structure of the ProcessRxSignalFunction. The existing case switch on the packet type (as shown in Figure 6-8) had just two cases (packet types) and a default case. Adding in the Start Sequence simply required the addition of a third START_TYPE packet. Similarly a START_SIGNAL and a START2_SIGNAL were added to the case switch on the signal name. The revised Nassi-Schneidermann Diagrams are given in Appendix 1.

6.8.2 Disconnection Test

The next test on this code involved disconnecting the serial cable between the PC's from one of the PC's. Both PC's were then run with both the main PLP and PLPTX code running and the Test task being used to simulate the required signals from the Device Manager. As expected, nothing happened until the serial cable was reconnected to complete the link and the "Touch and Find" process completed as normal.

The final test in this section was to utilise the presence of two serial ports on the desktop PC. The TTPCom "Mad Cow" Bluetooth EVB was connected to serial port "COM 2" and the serial cable for the "Touch and Find" process was connected to the other serial port "COM 1". The laptop (PC "A" – as shown in the test setup diagram in Figure 6-10) ran both the main PLP task and the PLPTX task and would use the TEST task to simulate the Device Manager. The desktop PC (PC "B") would run all of the TTPCom tasks and had the EVB connected. The serial cable was disconnected; "run" was pressed on both PC's and then after a brief period the serial cable was reconnected. The "Touch and Find" process completed as normal once the connection had been re-established showing that both the main PLP and PLPTX tasks were compatible with the existing TTPCom tasks.

6.9 CHAPTER SUMMARY

Chapter 6 has described the requirements, design, implementation and testing of the PLP Transport Task. The chapter started with a description of how the windows serial port works and how it should be set up for non-overlapped input and output. The text described the multi-thread nature of the serial interface and how "events" are used. The chapter then goes on to describe how the serial interface is used in the PLP Transport task and how the standard method for sending a signal out through the serial port from the PLPTX task was developed.

The signal structure design was explained together with how it was developed in order to simplify the data processing at the receiving end of the link. Finally the implementation and various tests that were carried out have been described, including the addition of the automatic start feature for the "Touch and Find" process.

CHAPTER 7 HARDWARE

The software tasks described in Chapters 5 and 6 were successfully implemented and then tested using a crossed over serial cable to provide the hardware link between the two devices to be paired. In this Chapter three types of hardware solution are investigated to provide the final link between the devices. The solutions developed are: - electrical contacts, infrared and inductive simple looping. Each of these methods of communication described has been designed to support full duplex communication as required by the Pairing Link Protocol. Another possible solution could be based on capacitive coupling.

The chapter starts by describing the first hardware concept developed which was the simple electrical contact solution. This solution uses a "hybrid" circuit to achieve full duplex communication using two contacts. The modifications to the software that were required in order to detect the "CONNECTED" state are explained together with the testing of the solution. The chapter goes on to describe the design of two possible infrared based solutions and the design, implementation and testing of the inductive loop solution.

7.1 SIMPLE ELECTRICAL CONTACT SOLUTION

The design of a full duplex simple electrical contact solution was complicated by the constraints placed on the design of the connectors by the need for all devices to have identical connectors. This "symmetry" is required because the "Touch and Find" system is to be used on devices which may need to pair to identical devices, thus a symmetrical connector is required. It is also necessary to know which of the contacts is the ground signal. One possible solution to this is to use contacts that are circular and concentric. It was decided that the use of three concentric circular contacts (as required for a traditional three wire full duplex serial link) would be both space consuming and unnecessarily complex and that a two contact solution would be best provided that a suitable circuit for providing full duplex over two wires could be designed.

The method proposed is essentially an electronic "hybrid" [53]. The "hybrid" transceiver allows the transmit and receive paths to be combined across a single line whilst having the ability to extract the data to be received from the combined path for the receive function. The "hybrid" is widely used in communications for separating go

and return transmissions; for example in every wired analogue 'phone. For the purposes of digital data transmission between two devices actually in contact, a simple realisation is possible, which might have the potential for low cost and to be largely realised in integrated form.

Figure 7-1 shows the circuit diagram of the proposed "hybrid" circuit. In each device, the data to be transmitted is buffered by two stages to provide Q and Q![12] signals. The Q signal is sent to the other device through R1 and applied to the input of a local Schmitt receive buffer through R2. The Q! signal is applied to the receive buffer through R3. The method of operation is simply that the resistor values are chosen so that, when connected, the local signal balances out at the input to the receive buffer. However the signal from the other side is not nulled out and hence is recovered and converted to a logic level by the buffer. It is assumed that the Tx buffers are both CMOS type which swing effectively from rail-to-rail; the rail voltages on each side are reasonably equal; and the Schmitt input thresholds are symmetrical around the mid-rail voltage.

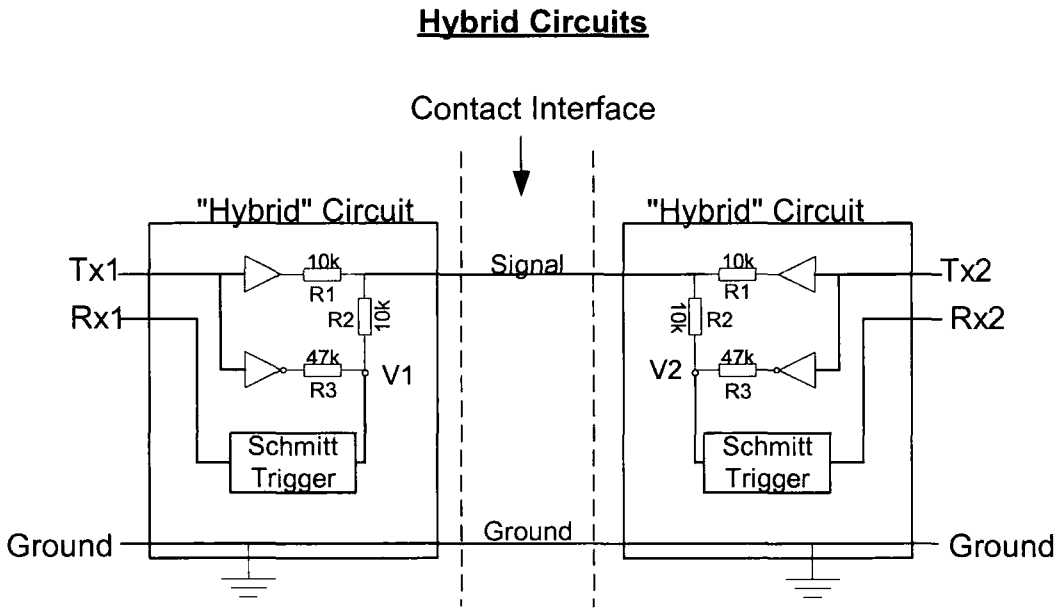


Figure 7-1 "Hybrid Circuit" Diagram

The circuit shown in Figure 7-1 was made up and tested to verify that it worked. A range of resistor combinations were used, but the values shown gave the best voltage

[12] ! means negated.

swing at "V1" and "V2", the inputs to the Schmitt trigger. The results of the testing of the "hybrid" circuit are shown in Table 7-1.

Input (Volts)		Output (Volts)	
TX1	TX2	V1	V2
5	5	3.50	3.50
5	0	2.06	2.94
0	5	2.94	2.06
0	0	1.51	1.51

Table 7-1 Table of Output voltages from basic circuit

The ability of the circuit to detect the "connected" state was investigated and it was found that due to the symmetrical nature of the circuit it would be impossible to determine whether or not another device was connected to the serial port. The "hybrid" circuit was tested to determine the voltage level at the input to the Schmitt trigger whilst the signal line was disconnected, i.e. the effect of TX1 on V1, which is the input to the Schmitt trigger. The results are shown in Table 7-2.

Input		"Connected"	"Disconnected"
TX1 (Volts)	TX2 (Volts)	Output (V1) (Volts)	Output (V1) (Volts)
5	5	3.50	3.49
5	0	2.06	3.49
0	5	2.94	1.51
0	0	1.51	1.51

Table 7-2 Table showing the effect of TX1 on V1 in the disconnected state.

The results shown in Table 7-2 show that with the existing circuit it is impossible to determine when the devices are in the "connected" state, as the voltage at "V1" is the same if TX1 is 5 volts in the "disconnected" state as it is when TX2 and TX1 are 5 Volts in the connected state. i.e it is impossible for the circuit to tell the difference between the "disconnected" state and the "connected" state when both inputs are the same.

Although various hardware solutions to detecting the "connected" state were considered, it was decided that it was best to minimise the complexity of the hardware and to use a software solution to ensure that signals that a device transmits are not received and processed by the same device. This is discussed in Section 7.1.2 "Detecting the Connected State."

7.1.1 Schmitt Trigger

The Schmitt trigger was used in the circuit to convert the signal received into a logic signal. It was not possible to use a Schmitt trigger IC (Integrated Circuit Chip) because the switching thresholds were not at an appropriate level. Thus it was necessary to construct a Schmitt trigger using a comparator in order to set the thresholds at the required levels. The Schmitt trigger used and its transfer characteristic are shown in Figure 7-2. The Schmitt trigger's thresholds were designed to be in between the voltages received (2.06V and 2.94V) when the inputs were not the same, i.e Tx1 was 5V and Tx2 was 0 V and vice versa. The Schmitt trigger switches at 2.25V and 2.80V.

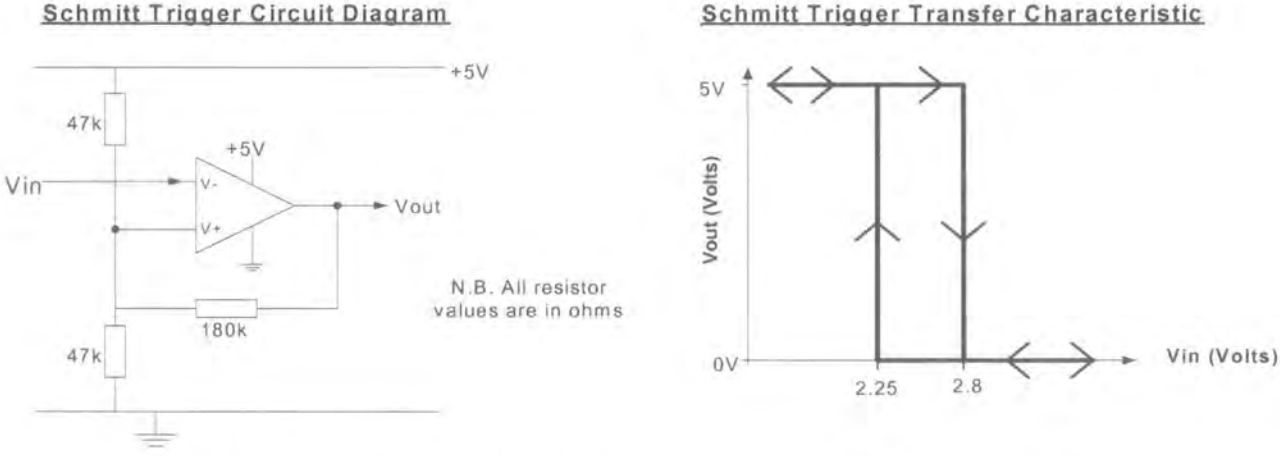


Figure 7-2 Schmitt Circuit Diagram and Transfer Characteristic

The Schmitt trigger shown above in is an inverting Schmitt Trigger, so an inverter has been added after the Schmitt trigger. The Maxim chip was added as a voltage converter between the input/output of the PC (which uses RS232 compliant voltages: +12V, -12V) and the "hybrid" transceiver circuit (requires CMOS voltage levels +5V, 0V). A block diagram of the circuit is shown in Figure 7-3. The full simple electrical contact solution is shown in Figure 7-4.

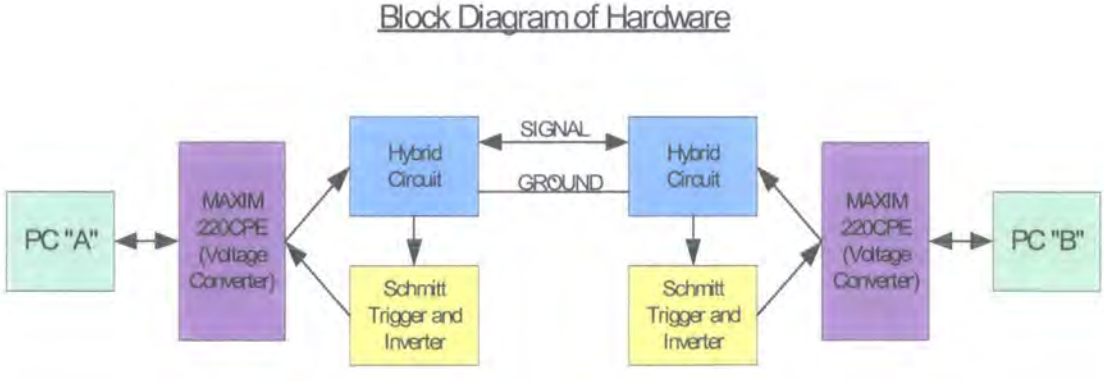


Figure 7-3 Block Diagram of Hardware

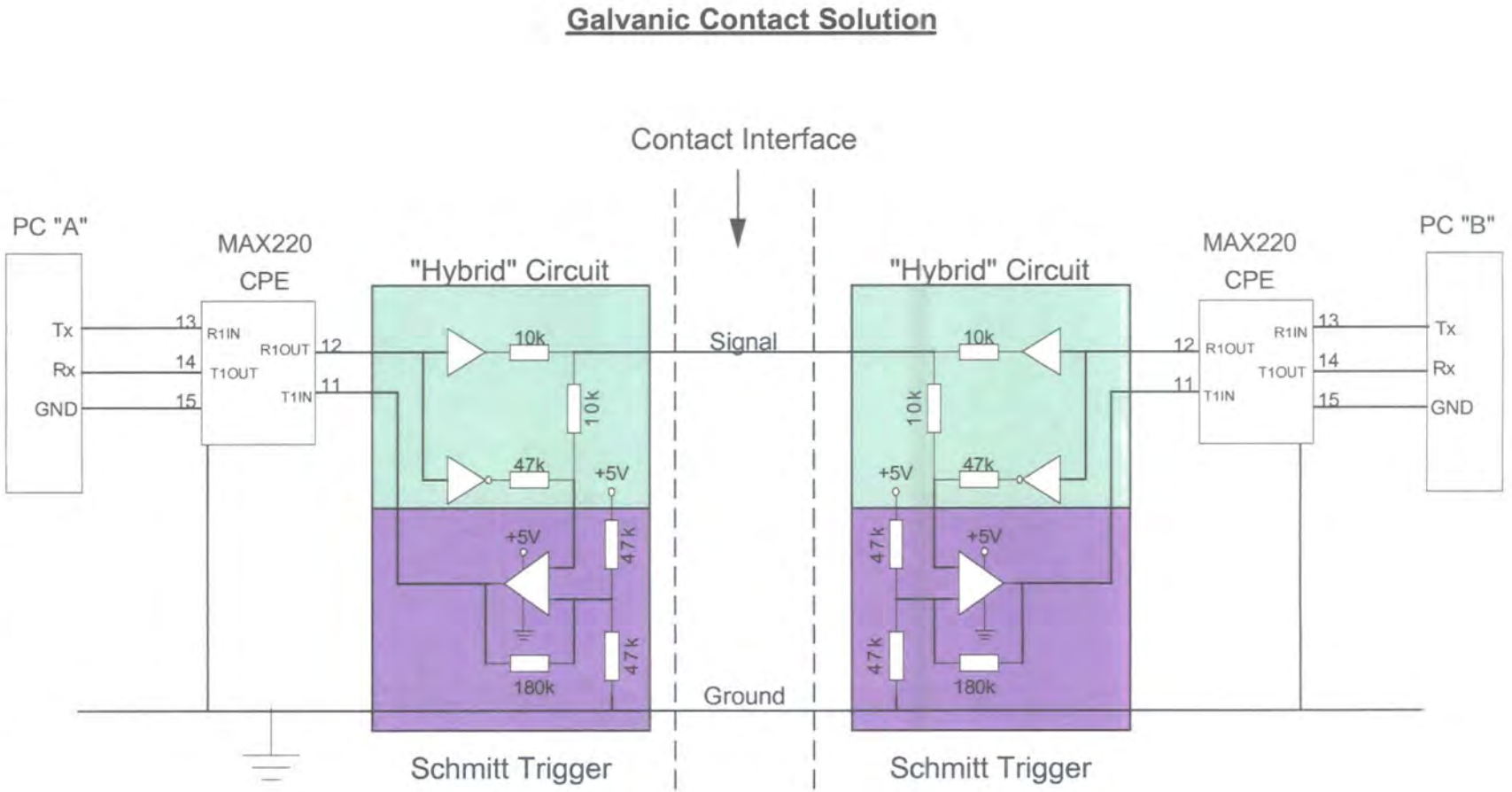


Figure 7-4 Full Simple Electrical Contact Solution.

NB. MAX220 chip shows Interface connections only.

Having constructed the "hybrid" circuit, Schmitt Trigger and Inverter, the circuit was tested using two single pole double throw switches to provide the inputs. The Schmitt triggers were also tested at a frequency of 1MHz, using a square wave input from a signal generator to verify that they were capable of switching at the required rate.

The next stage was to connect the PC's to the circuits. This was done using a serial connector into the PC serial port with wires connected to the Tx, Rx and Ground lines that were then fed into the Maxim chip and "hybrid" circuit as required. The signal line and ground between the sets of hardware for each device were also connected and then the main PLP and PLPTX code was run on both machines with the Test task running to simulate the required signals from the Device Manager. The local and remote device information was correctly returned to the test task (which was being used as the calling application) as defined in the "Pairing Link Protocol". The "Touch and Find" system worked as intended with no problems encountered in the testing of the combined hardware and software.

7.1.2 Detecting the Connected State.

As indicated previously, a method is required to detect the "connected" state. Without some form of software changes the system was incapable of recognising when another device was connected; it would simply send signals to itself and then process them. This would result in the "Touch and Find" process completing incorrectly as the only device information would be that of the local device which would fill both the remote and local device information areas.

The simplest solution appeared to be to add a unique header to each packet sent. The header would consist of three known start bits (so that the start of a packet could be easily detected) and would then be followed by a unique device identification number. The full Bluetooth address of the transmitting device was used as the unique device identification number. The PLPTX task would then compare the received device Identification (device ID) with that of the local device. If the two ID's are the same then the PLPTX task would throw away the received signal; if different, the received signal is processed in the normal way. This allows the software to detect the "connected" state.

The header would be included on all packets sent and if at any stage the device ID of the received signal became the same as that of the local device, it would be known that the connection had been lost and the signal would be thrown away. This solution would also be compatible for use with both the inductive and infrared solutions. The

signal structure (shown in Figure 7-5) now has an additional 3 byte start sequence and a 6-byte unique device ID (Bluetooth address).

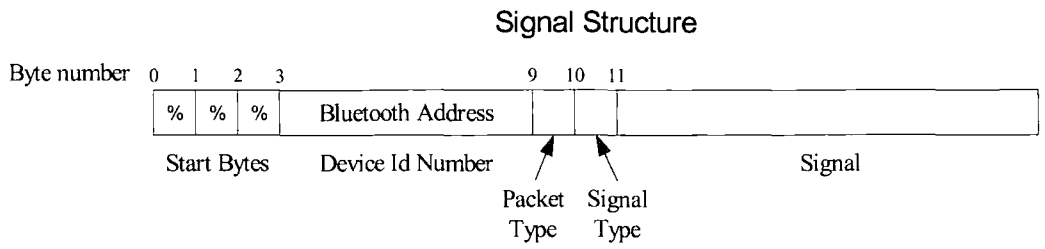


Figure 7-5 Generic Signal Structure

The implementation of the above change was simple as it was compatible with the existing structure in both the rx and tx State switches – it simply required adding two extra states to the rxStates. The changes required have been represented in Figure 7-6 and in the Nassi-Schneiderman Diagrams in Appendix 2.

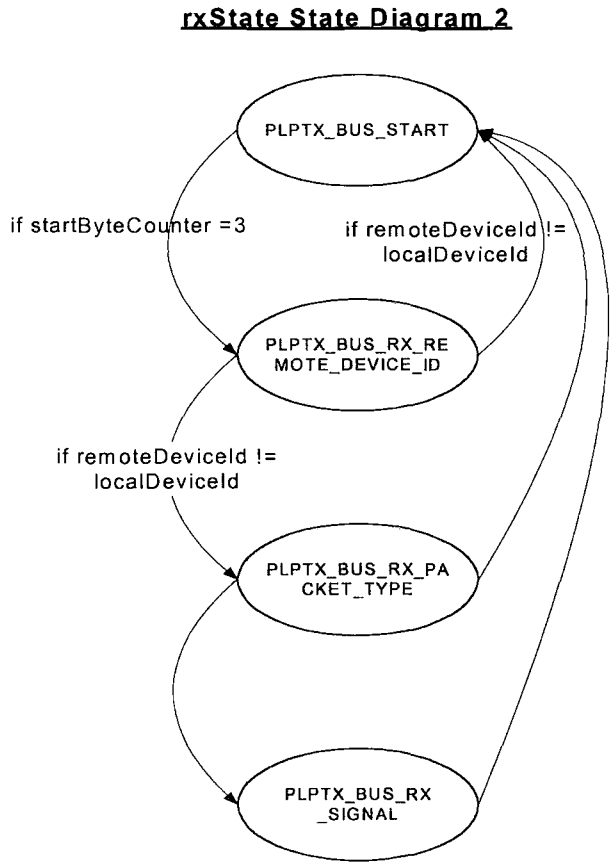


Figure 7-6 RxState Diagram 2

In the process of implementing these changes, the system was made more robust so that instead of failing if an incorrect signal was received, it would simply throw away the received signal. The final signal flow diagram used in the "Touch and Find" system is shown in Figure 7-7.

Pairing Link Protocol Signal Flow

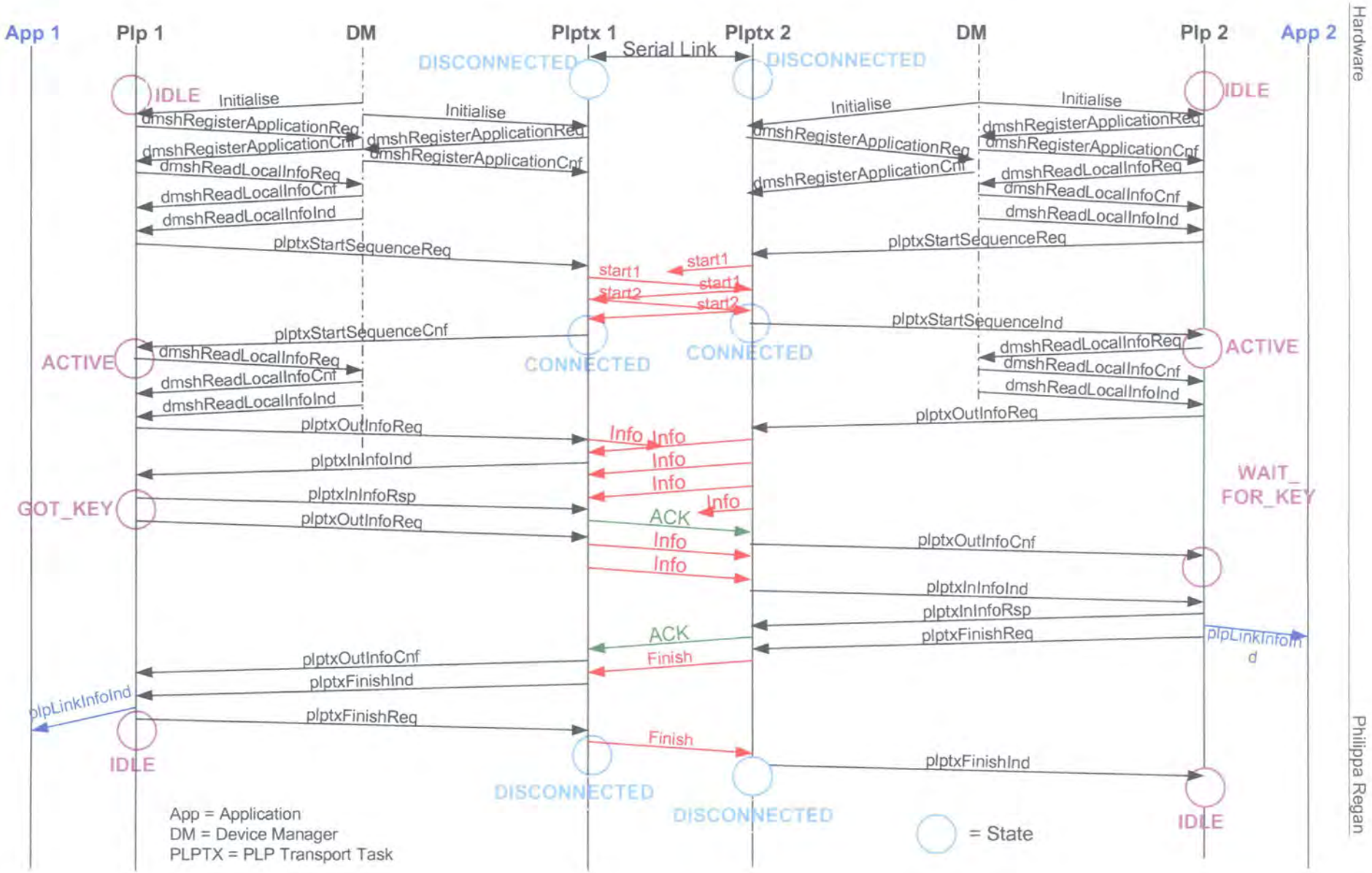


Figure 7-7 Final Signal Flow Diagram

7.1.3 Physical Contacts

The physical design of a set of contacts suitable for use in the simple electrical contact solution for the "Touch and find" system was considered. Due to the requirement to have exactly the same contact on each device and the need to have a signal Ground in a known place, the only solutions available were concentric contacts. A possible arrangement is shown in Figure 7-8. It may be necessary to spring load the contacts in order to ensure a good connection. Although it would be possible to short circuit the contacts if they were misaligned or a coin came into contact with them, the output resistance of the circuit is sufficiently high that this would not matter.

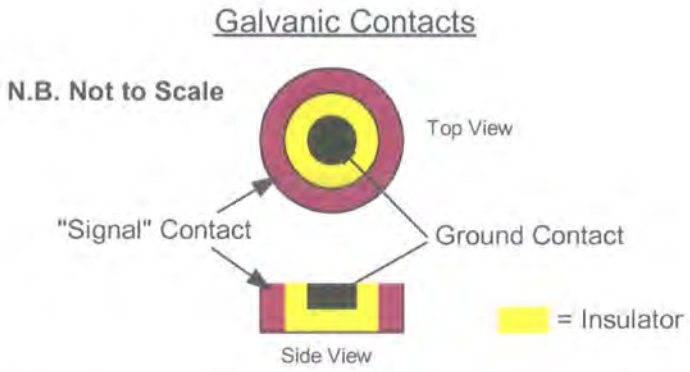


Figure 7-8 Concept Diagram of Simple Electrical Contacts

7.1.4 Testing the Simple Electrical Contact Solution

The simple electrical contact solution worked well when tested using two wires (signal and ground) to connect the two devices. As one of the PC's being used to test the devices only had one serial port, it was necessary to simulate the role of the Device Manager using the Genie Test task. The local device information that is normally provided by the device manager was sent manually by the user from the Genie Test Task. The following tests were carried out successfully:-

1. With both signal and ground connected, the software was run through Genie on both systems (PC "A" and PC "B" in Figure 6-10 PLPTX Test Setup).
2. With just ground connected, "Run" was pressed on Genie on both systems. Nothing happened until the "signal" line was connected and then the process completed successfully. This showed that the "Connected" state had been successfully detected.
3. The test outlined in 2 was repeated but this time the "signal" line was quickly disconnected and reconnected in the middle of the process. The process

completed with the correct data after extra signals had been sent to cope with the signal loss. This demonstrated the robustness of the code.

Finally, the TTPCom "Mad Cow" Bluetooth Evaluation Board was connected to the second serial port on PC B, so that the system could be tested with the device manager operating on one side. Again the process completed successfully showing that the complete "Touch and Find" system had been implemented successfully using the "simple electrical contacts" hardware solution as the physical medium.

7.2 INFRARED SOLUTION

Two concepts were used for the development of an Infrared solution. The first was based on a simple pair of matched infrared transmitters and detectors and the second was based on an IrDA (Infrared Data Association) chipset. Due to time constraints neither solution was constructed, particularly as the IrDA chipset would require a PCB to be designed and made. However, designs based on both techniques were developed and analysed here.

A simple and low cost solution is shown in Figure 7-9. It is based on using amplitude modulation of an Infrared carrier (generated locally) with the data signal. This is then transmitted through an Infrared transmitter which is frequency matched with the Infrared receiver used in the other device. The received signal is passed through a low pass filter to remove the high frequency component, leaving just the data signal, and is then input to an amplifier. The gain of the amplifier was to be determined according to the amplitude of the received signal. A Maxim level converter was required between the PC serial port and the circuit to change the voltage levels between RS232 and CMOS. Infrared modem IC's (Integrated circuits) were researched but there seemed to be a lack of "non IrDA" infrared modems available. Thus the solution utilising the IrDA chipset was realised, this is shown in Figure 7-11.

The IrDa compliant chipset design was based on the data sheets and application notes of the chips used [54] [55]. The IrDA solution should work without significant changes as it is designed to be used as a cable replacement for use between PC's. No software changes should be required.

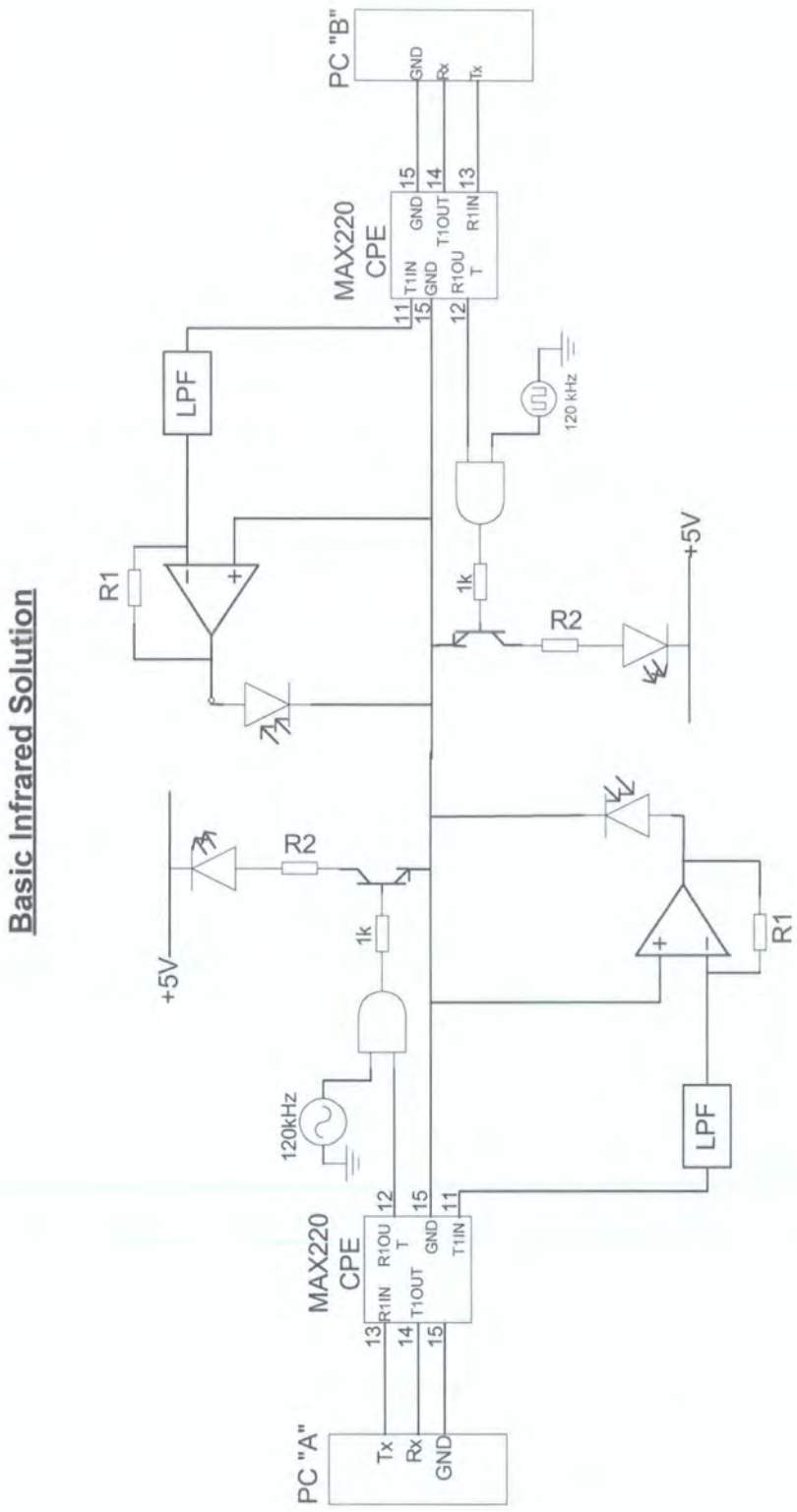


Figure 7-9 Basic Infrared Solution

It should be noted that the interface between the two devices is different to that shown in Figure 5-1 as there is no longer a ground signal. The interface between the PLPTX Transport tasks for the Infrared solutions is shown in Figure 7-10.

Interface between devices for Infared solutions

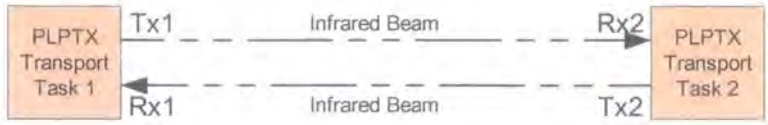


Figure 7-10 PLPTX Interface for Infrared solutions

The IrDA solution shown in Figure 7-11 uses the Maxim level converter to interface between the PC and the circuitry. The HSDL7001 chip receives a serial data input (at CMOS voltage levels) from the Maxim level converter and modulates and demodulates signals to and from the HSDL1001 IrDA transceiver. The HSDL7001 is compliant with the IrDA 1.0 physical layer specification and has an external 3.6864MHz as it is run in the internal clock mode. The transceiver operates at 875nm.

The infrared solutions offer a good alternative solution to both the simple electrical contact solution and the inductive looping solution. The link is secure, (although not as secure as the electrical contact solution that requires physical contact) as Infrared is restricted by line of sight and typically works within a 30° cone. Both the basic Infrared solution and the IrDA solution are low power, making the Infrared solutions suitable for use in mobile devices. In addition, Infrared communication is a mature technology, is widely available and is very reliable.

The IrDA solution has the advantage that it is almost ready made and very reliable. However it is also a rather expensive solution if the IrDA chipset is only used for Bluetooth Pairing. Ideally a full IrDA port would be implemented in the device and if integrated would also allow Bluetooth pairing, but this would be difficult as the IrDA stack is very complex. The best long term solution (unless the device has an IrDA port) is to use the basic IrDA solution which is simpler and lower cost.

IrDA Solution

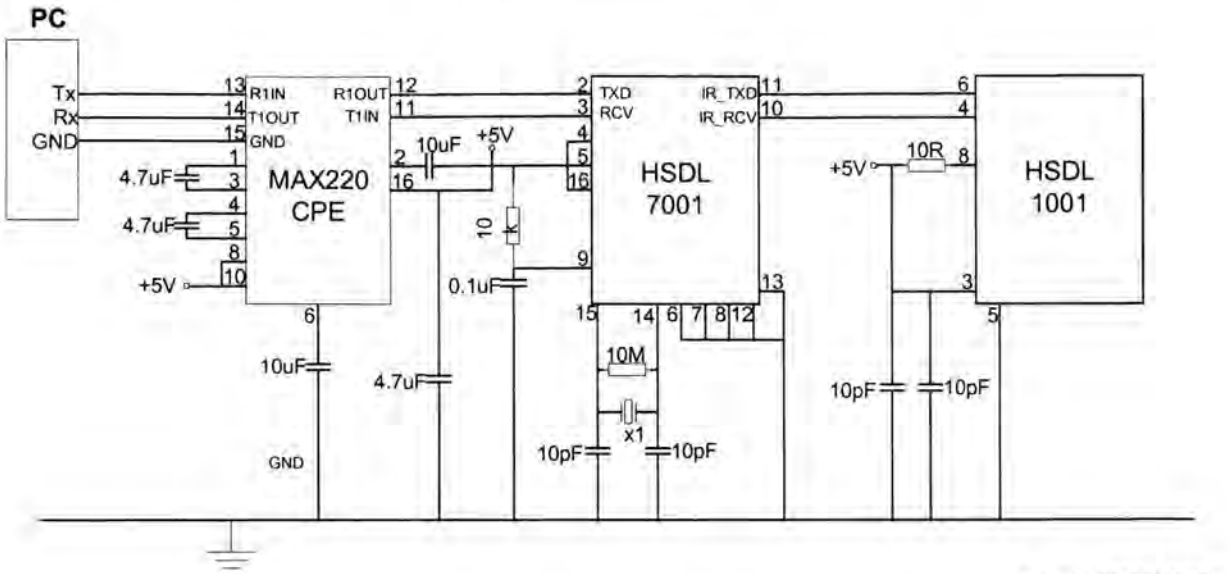


Figure 7-11 Circuit Diagram of IrDA Solution

7.3 Inductive Loop Solution

The concept behind this solution is to have two coils, one from each device, placed in close proximity to one another. One device will then transmit an ac (alternating current) signal on the first coil and by means of the induced voltage in the second coil, the second device can detect the transmitted signal. Ideally a circuit enabling full duplex transmission using a single pair of coils was to be developed.

Two coils with an inductance of 1mH were made up and subjected to some basic tests in order to facilitate the design process. The inductance and self-resonant frequency of the coils was measured using an impedance analyser. The self-resonant frequency of the coils was determined to be outside the range of frequencies that might be used for modulation

In order to determine if modulation would be required a signal generator was used to produce a signal with a frequency equal to the minimum frequency that would be output from the RS232 serial link. It was determined that modulation would be required as the signal did not vary quickly enough to be received on the second coil in a recognisable form.

Having determined that modulation was required, the next step was to determine if resonant circuits would be beneficial. The circuit shown in Figure 7-12 was used to evaluate the amplitude of the signal received.

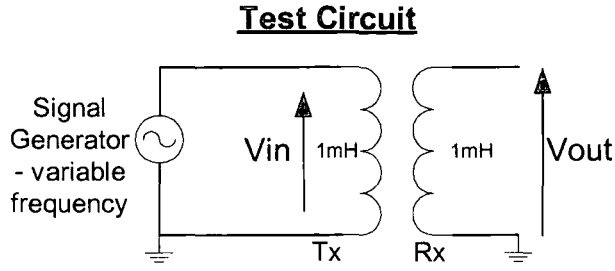


Figure 7-12 Circuit Diagram of Test Circuit

For a 5V peak-peak input signal, the output voltage "Vout" on the above circuit was 1V peak-peak. The received signal did not have a sufficiently large amplitude to process the received signal reliably whilst allowing for variations in amplitude due to the distance between the coils and noise. Thus some form of resonant circuit would be required.

Using the inductor's self-resonant frequency was considered but determined to be unsuitable as each circuit's resonant frequency is different and it would be extremely difficult to create coils with the same resonant frequency.

A full duplex inductive solution based on the "hybrid" circuit used in the simple electrical contact solution was investigated, but no solution found. There were many factors that complicated the development of the required circuit.

- The circuit must resonate in order to create a sufficiently large voltage on the receive coil.
- Modulation was required.
- Transmit and receive must be carried out on one circuit. It was not possible to have a serial "RCL" network for the transmit side and a parallel "RCL" network for the receive side.
- The inductance of the coil and hence the resonant frequency were affected by the distance separating the two coils.
- The distance between the two coils affected the inductive coupling.
- Designing a physical aligning device that could be used to align the two coils and fix the distance between them was very difficult as it is necessary for the devices to be identical.

The above factors made it very difficult to develop a full duplex circuit with two coils in the limited time available. Instead, a simpler solution was investigated.

7.4 Four Coil Solution

The use of four coils greatly simplified the task. It would be possible to have separate transmit and receive coils, thus enabling optimisation of the transmit and receive circuits by using a series "RCL" network for the transmit circuit and a parallel "RCL" network for the receive circuit.

The use of two sets of linked coils lends itself to each linked pair of coils being concentric – for example using a small coil as a transmit coil inside the larger receive coil. Each device would have both a large coil around a recess and a small prominent coil. This arrangement would increase the linkage and should enable power transfer across the inductive link if required (as in electric toothbrushes). The envisaged connector is shown in Figure 7-13.

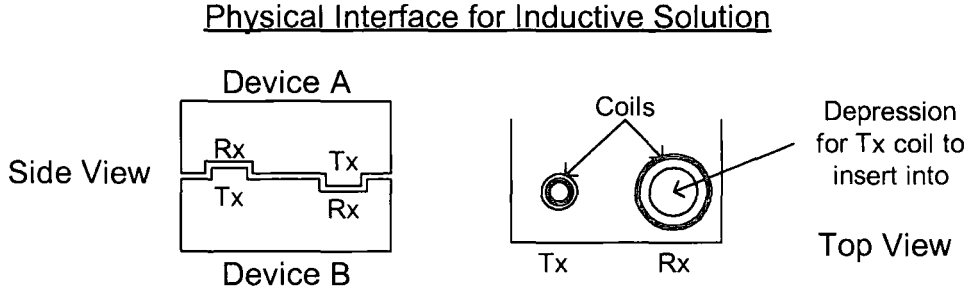


Figure 7-13 Physical Interface For Inductive Solution

For test purposes four 1mH coils were made up. The two larger coils had an internal diameter of 19mm and the two smaller coils had an external diameter of 15mm. Figure 7-14 shows the arrangement of the Transmit (Tx) and Receive (Rx) coils and their respective sizes.

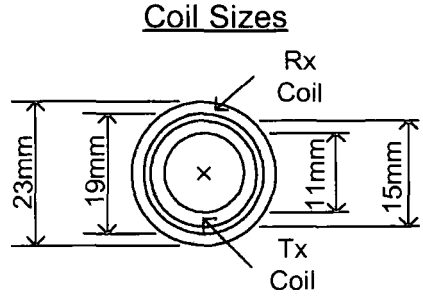


Figure 7-14 Inductor coil Alignment

The number of turns required was estimated using Equation 1.

$$N = \sqrt{\frac{L_{\mu H}(6a + 9h + 10b)}{(0.31a^2)}}$$ Equation 1 [55]

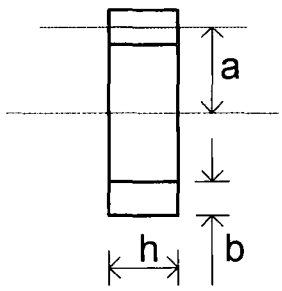


Figure 7-15 Key to Equation 1

Where : -

$L_{\mu H}$ = Inductance in micro Henry's.

a = average radius of coil in cm.

b = winding thickness in cm.

h = winding height in cm.

N = number of turns.

The coils were made up and the impedance of the coils was measured using an Impedance Analyser. It was verified that the self-resonant frequency of the coil was not in the frequency range to be used.

7.4.1 Calculating the Resonant Frequency of the Circuit.

$$\varpi_{res} = \frac{1}{\sqrt{LC}}$$ **Equation 2**

$$\omega_{res} = 2\pi f_{res}$$ **Equation 3**

Combining Equations 2 and 3 gives

$$f_{res} = \frac{1}{2\pi\sqrt{LC}}$$ **Equation 4**

It was decided to use a modulating frequency of approximately 200KHz requiring a capacitance of 680pF to be used (to the nearest available capacitance value). The inductance and capacitance values were then substituted into Equation 4 as shown below.

$$L = 1\mu H$$
$$C = 680pF$$
$$f_{res} = \frac{1}{2\pi\sqrt{1\times10^{-3}\times680\times10^{-12}}}$$
$$f_{res} = 193kHz$$

Thus the predicted resonant frequency of the resonating circuit is 193kHz. This was then verified experimentally using the circuit shown in Figure 7-16. The test circuit showed that the two circuits (transmit and receive) did not resonate at exactly the same frequency and that although the amplitude of the received signal could be improved through tuning, it was not necessary in order to demonstrate that the circuit would work. The frequency used throughout the development of the inductive looping circuit was 193kHz, as this was the frequency at which the voltage across the secondary (receive) coil was maximised.

Resonant Frequency Test Circuit

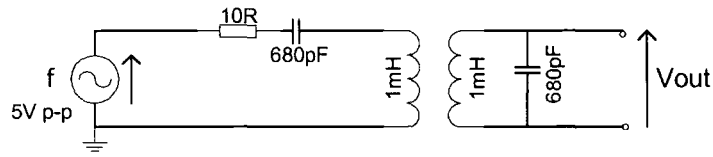


Figure 7-16 Resonant Frequency Test Circuit

Having determined the frequency of the modulating signal it was simply a matter of modulating and demodulating the signal. Modulation of the 193kHz signal was carried out by the use of a transistor as this allowed a bipolar input. When the data bit was a "1", the modulating signal would be applied across the coil and when the data bit was a "0", the coil would be short-circuited, thus providing 100% amplitude modulation. The circuit diagram of the circuit used during the development process is shown in Figure 7-17.

The signal applied to the coil, when the data bit was "1", was not a perfect sine wave due to the transistor not being turned on until the voltage at the base reached approximately 0.6V, however the imperfect sine wave did not cause any problems.

Inductive Solution Development Circuit Diagram

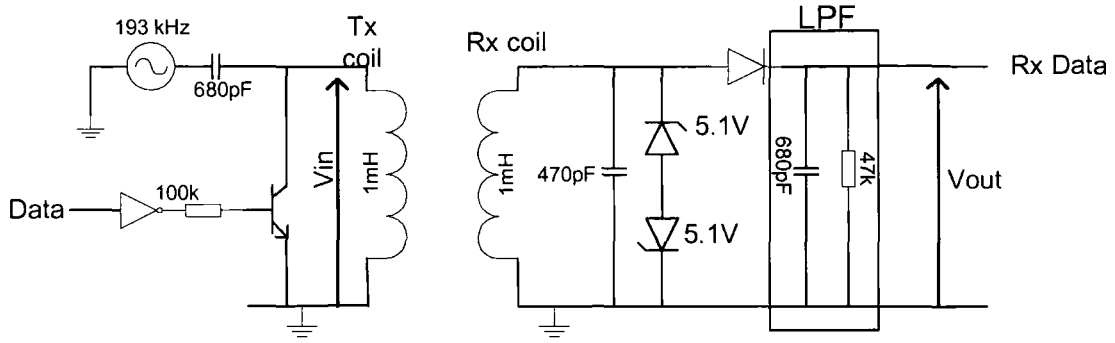


Figure 7-17 Inductive Solution Development Circuit Diagram

The receive circuit was slightly more complex. Firstly the voltage across the inductor was measured to be up to 39.2 V peak-peak and hence required limiting by means of a pair of 5.1v Zener Diodes. The signal was then rectified with a diode and demodulated by passing it through a Low Pass Filter.

The low pass filter was designed to ensure that the capacitor smoothed the output sufficiently to be read as a "1" whilst the data bit was a "1" and such that the voltage fall off was sufficiently fast that when the data changed to a "0" the received signal was also a "0". The time taken for the voltage to drop to 2.5V, following the data bit going low, was 12.8 µS which, as required, is considerably shorter than the bit time. A plot of Voltage against time in shown in Figure 7-18. The points on the circuit at which the voltages were measured are shown in Figure 7-18.

Finally it was necessary to clean up the signal with a Schmitt trigger so that a clean logic signal could be input into the Maxim Voltage converter for transmission to the PC via an RS232 cable. The Inverting Schmitt trigger was designed to have thresholds at 2.25V and 2.8V. The complete circuit is shown in Figure 7-19.

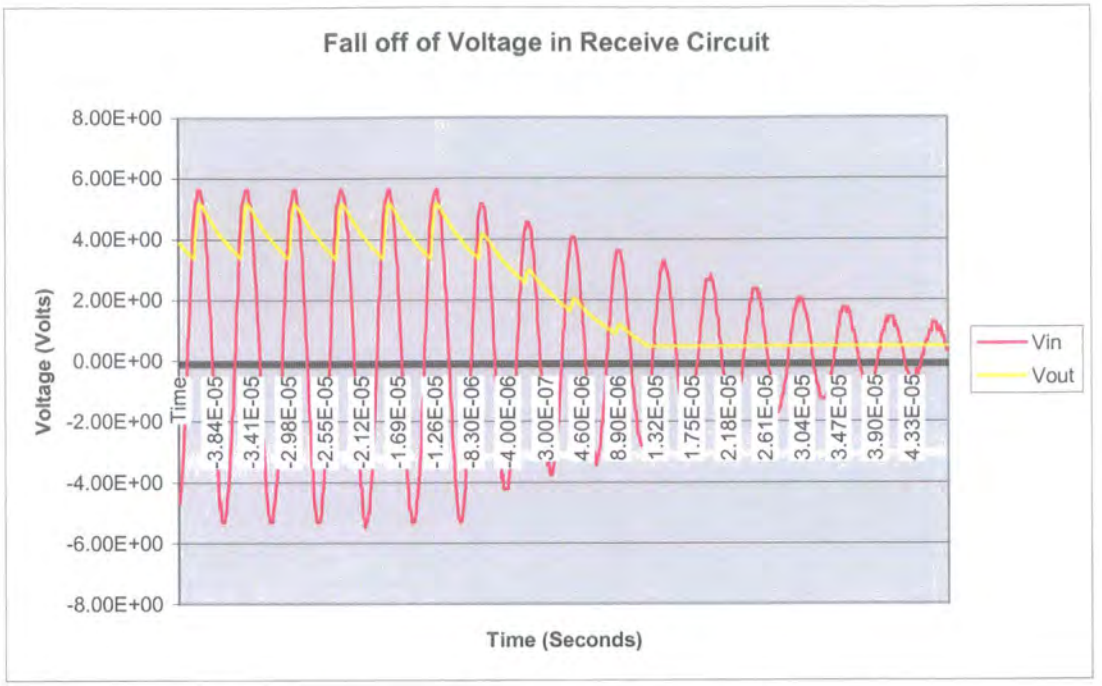


Figure 7-18 Fall-off of Voltage in Receive Circuit

Inductive Solution

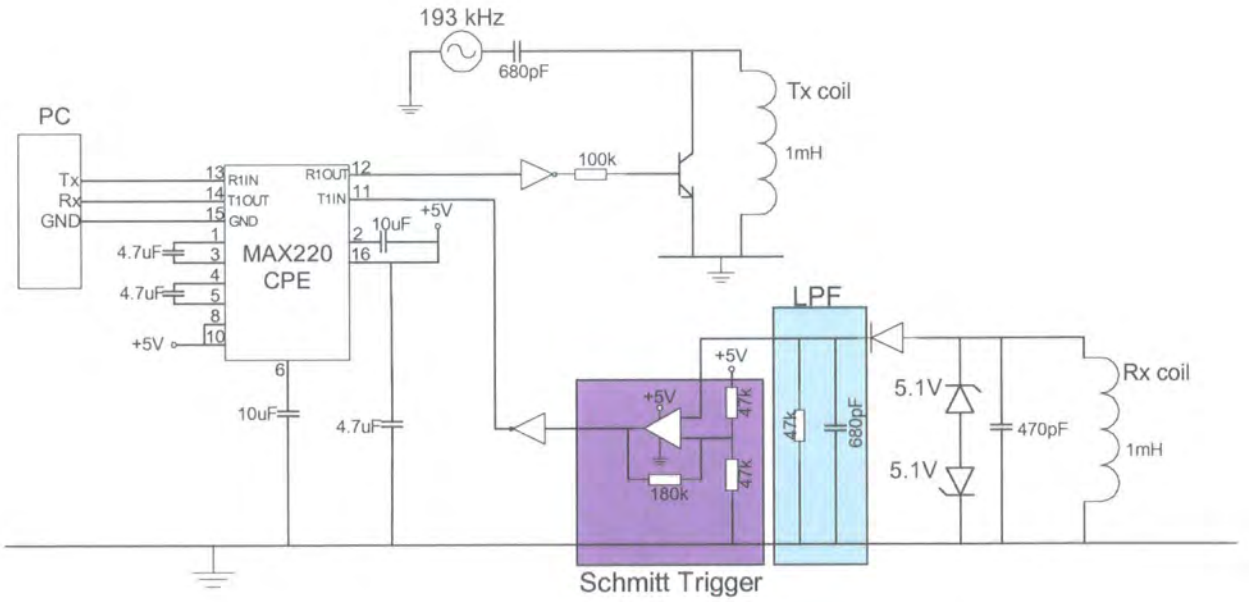


Figure 7-19 Circuit Diagram of Inductive Solution

7.4.2 Testing the "Four Coil" Inductive Solution

The circuit shown in Figure 7-19 was constructed twice, once for each device. Finally the system was tested by running the complete "Touch and Find" code through Genie. Both the main PLP task and the PLPTX task were run, together with the Test task (to simulate the Device Manager). The following tests were carried out successfully: -

1. With both sets of coils aligned, the software was run through Genie on both systems. The process completed successfully.
2. With both coils apart, "Run" was pressed on Genie on both systems. Nothing happened until the coils were brought into alignment and then the process completed as normal.
3. The TTPCom "Mad Cow" Bluetooth Evaluation Board was connected to the second serial port on PC B, so that the system could be tested with the device manager operating on one side. The "Touch and Find" process completed successfully.

This showed that the "Touch and Find" system worked well using the four coil inductive solution. None of the code used with the simple electrical solution needed to be changed when the physical medium that the "Touch and Find" system was using was changed, showing that the Pairing Link Protocol designed was suitable for use with more than one hardware solution.

7.5 CHAPTER SUMMARY

Chapter 6 introduced the three types of hardware solution developed. The development of the simple electrical contact solution and the software changes required in order to detect the "connected" state have been described. The "Touch and Find" system is then tested with the simple electrical contacts solution being used as the physical connection. The design and evaluation for two different Infrared solutions has been documented and a four coil Inductive Coupling solution has been thoroughly investigated, designed and tested with consideration given to the physical interface design.

This Chapter shows how flexible both the "Touch and Find" software and the Pairing Link Protocol are. The "Touch and Find" system has been shown to work very well using a simple electrical contact solution and an Inductive Coupling solution. Two Infrared solutions have also been designed further extending the hardware options for use with the "Touch and Find" system.

CHAPTER 8 CONCLUSIONS AND SUMMARIES

This chapter covers the evaluation of the hardware/software and of the complete system. The thesis is then summarised and conclusions are drawn before suggestions are made for further work on the "Touch and Find" system. At the end there is a concluding statement.

8.1 EVALUATION

The "Touch and Find" system was developed to improve the usability of Bluetooth pairing in devices. The system was also developed to facilitate simple Bluetooth pairing when using a PAN Gateway or other device with no user interface. It was designed to enable the information required for Bluetooth pairing to be exchanged by touching the two devices together momentarily. It was hoped that various different physical mediums could be used to transfer the information for the "Touch and Find" system.

The Evaluation section was been split into a hardware and software section which is then followed by an evaluation of the complete system.

8.1.1 Software and Hardware

The Software has been implemented successfully in the form of two tasks; the main PLP task and the PLP transport task. The software allows a successful demonstration of the concepts of the "Touch and Find" system. Modularisation has been achieved and the lower level transport tasks have been carried out in the PLPTX task, making it simpler to implement on other platforms. Throughout the development there have been some modifications required to the code and to the Pairing Link Protocol, but the resulting software supports a robust serial link pairing mechanism, fulfilling the aims of the "Touch and Find" system. Both the main PLP task and the PLPTX task satisfy the requirements set out in sections 5.3 and 6.1.

The main PLP task is initiated from a higher level application (as required), it adheres to the Pairing Link Protocol designed and satisfies the interface requirements of the lower layers and the Device manager. The main PLP task does not block the processor allowing other Bluetooth tasks to be completed as normal. The PLP task successfully interacts with the device manager as required and is independent of the physical medium used to provide the serial link between the devices.

The PLPTX (Transport) task developed is independent of the physical medium used. It uses the Windows serial port for non-blocking data I/O and generates suitable signals for transmission across the physical layer. The transport task is also responsible for receiving and processing signals and carries out low level tasks concerned with signal transport across the physical layer. The PLPTX task works with different physical mediums as required.

The hardware developed works well with the "Touch and Find" software, allowing successful demonstration of the feasibility of the proposed "Touch and Find" system. All of the proposed hardware solutions provide an intuitive pairing method.

The simple electrical contact "hybrid" solution (see section 7.1) showed that a full duplex contact solution based on two wires was feasible. One potential problem for this "hybrid" simple electrical contact solution is the contacts becoming corroded/rusty leading to a bad connection. It is likely that the contacts will need to be mounted on a spring system so that when two devices are touched together a good electrical contact is formed. However this results in a potential mechanical weakness. The simple electrical contact solution is secure as physical contact is required. This solution is also relatively simple and cheap to implement. One possible problem area is the risk of the exposed contacts being damaged.

The "hybrid" circuit did not cancel out the local signal at the input to the receive buffer as well as expected and so a formal analysis of the circuit was carried out; this is shown in section 8.1.2.

8.1.2 Hybrid Circuit Analysis

For the purposes of this analysis the following definitions are made: -

"High" end is defined as Tx+ = V and Tx- = 0.

"Low" end is defined as Tx+ = 0 and Tx- = V

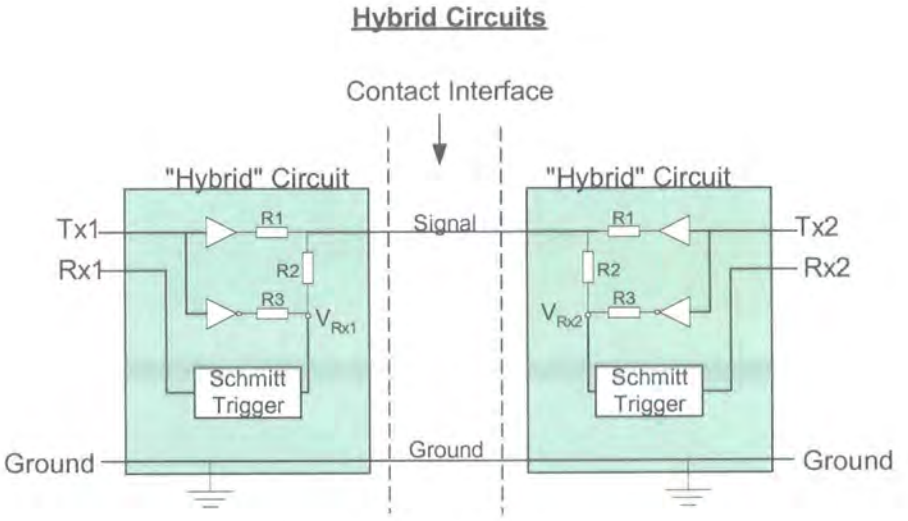
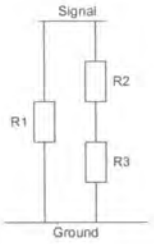


Figure 8-1 "Hybrid" Circuit for Analysis

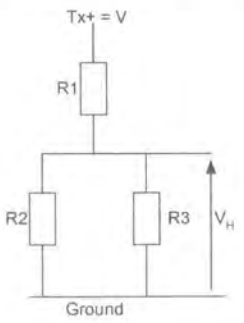
The impedance of one side of the resistor network is given by: -



$$R_1 // (R_2 + R_3) = \frac{R_1(R_2 + R_3)}{(R_1 + R_2 + R_3)} = R_0$$ **Equation 5**

Where R_0 is the Thevenin resistance.

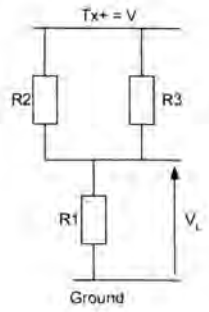
The Thevenin equivalent of the "high" end is given by: -



$$V_H = \frac{V(R_2 + R_3)}{(R_1 + R_2 + R_3)}$$ **Equation 6**

V_H in series with the Thevenin Resistance R_0.

The Thevenin equivalent of the "low" end is given by: -



$$V_L = \frac{VR_1}{(R_1 + R_2 + R_3)}$$ **Equation 7**

V_H in series with the Thevenin Resistance R_0.

If both ends are high: -

The voltage on the "signal" line is V_H and the voltage at the input to both Schmitt triggers (V_{Rx1} and V_{Rx2}) is: -

$$V_{Rx1} = V_{Rx2} = \frac{V_H R_3}{(R_2 + R_3)} = \frac{VR_3}{(R_1 + R_2 + R_3)}$$ **Equation 8**

If one end is high and the other is low (Tx1 = high, Tx2 = low), the "high" end presents a voltage of V_H through R_0 and the "low" end presents a voltage of V_L through R_0. Thus the voltage on the signal line is given by: -

$$\text{Signal} = \frac{V_H + V_L}{2} = \frac{V}{2}$$ **Equation 9**

The voltage at the input to the "low" end Schmitt trigger V_{Rx2} is given by the sum of the signal voltage and the potential divider applied to Tx-, as shown below: -

$$V_{Rx2} = \frac{V}{2} + \left[\frac{(V - \frac{V}{2})R_2}{(R_2 + R_3)}\right]$$

$$V_{Rx2} = \left(\frac{V}{2}\right)\left(1 + \frac{R_2}{(R_2 + R_3)}\right)$$ **Equation 10**

In order to be "deaf" to the local signal at the input to the Schmitt trigger, we require Equation 8 and Equation 10 to be equal: -

$$\frac{VR_3}{(R_1+R_2+R_3)} = \left(\frac{V}{2}\right)\left[1+\frac{R_2}{(R_2+R_3)}\right]$$ Equation 8 = Equation 10

$$R_3 = \frac{(R_1+R_2+R_3)}{2}\left[1+\frac{R_2}{(R_2+R_3)}\right]$$

$$R3 = \frac{(R1+R2+R3)}{2} + \frac{R2(R1+R2+R3)}{2(R2+R3)}$$

$$R_3 = \frac{(R_1+R_2+R_3)(R_2+R_3)+R_2(R_1+R_2+R_3)}{2(R_2+R_3)}$$

$$0 = \frac{(R_1+R_2+R_3)(R_2+R_3)+R_2(R_1+R_2+R_3)-2R_3(R_2+R_3)}{2(R_2+R_3)}$$

$$0 = R_1R_2 + R_1R_3 + R_2^{\,2} + R_2R_3 + R_2R_3 + R_3^{\,2} + R_1R_2 + R_2^{\,2} + R_2R_3 - 2R_2R_3 - 2R_3^{\,2}$$

$$0 = -2R_1R_2 - R_1R_3 - R_2R_3 - 2R_2^{\,2} + R_3^{\,2}$$

$$0 = R_3^{\,2} - R_3(R_1+R_2) - 2R_2(R_1+R_2)$$ **Equation 11**

Setting $R_1 = R_2 = r$ and substituting into Equation 11 gives: -

$$R_3^{\,2} - 2rR_3 - 4r^2 = 0$$

Solving the quadratic equation yields: -

$$R3 = 3.236r$$ **Equation 12**

Equation 12 shows that given that R1 = R2 = 10kΩ, R3 (see Figure 8-1) should have been a 32kΩ resistor instead of a 47kΩ resistor. This was due to an incorrect earlier analysis of the "hybrid" circuit. The use of a 47kΩ resistor instead of a 32kΩ did not stop the circuit working, but it would have made the switching thresholds on the Schmitt trigger input closer together than would otherwise have been necessary.

8.1.3 Other Hardware Solutions

The Infrared solution proposed was not implemented but both the proposed Infrared links are reliable and will not result in any software changes. The IrDA solution remains a rather expensive option at present, although this would be reduced if an IrDa port was already needed in the device and the necessary interface for the "Touch and Find" system could be incorporated. The Infrared solutions provide a secure means of Bluetooth pairing as Infrared communication is restricted by line-of-sight and only works in approximately a 30° cone. The Basic Infrared solution developed is a cheap, simple, reliable circuit. Both the Basic Infrared solution and the IrDA solution are low power and hence suitable for use in mobile devices.

The Inductive solution provides a particularly elegant solution to the problem, as it requires nothing to be mounted on the surface of the device. The "four coil" solution used works well, giving a large amplitude signal at the receiver. The sizes of the coils have been designed to give good coupling and to enable a self-alignment mechanism that will increase the reliability of the system. The Inductive solution could also potentially be optimised to allow battery charging via the same coils in a similar manner to that used in modern electric toothbrushes.

8.1.4 System

The system tests have shown that it is possible to transfer the required data for Bluetooth pairing across a serial link, in which the physical medium is either a simple electrical contact link or an inductive link. Infrared solutions have also been designed, but not implemented. The tests showed that the required data could be transferred very quickly and thus that it would be easy for the user to manually hold the connection together whilst the process completed. Unfortunately no "start to finish" time for the process to complete was recorded, as only one of the PC's used had two serial ports. Thus the system was dependent on a user pressing a button to activate a script to simulate the data arriving from the Bluetooth Device Manager.

Throughout the implementation of the software and hardware, the original Pairing Link Protocol was modified and enhanced. The successful completion of the "Touch and Find" process, using two different physical mediums and subjected to various tests (outlined in earlier sections) show that the Pairing Link Protocol was suitable for use with a variety of mediums and that it was sufficiently robust to create a reliable link. The decision to use full duplex communication across the serial link and to use a "broadcast" type protocol has been shown to result in a robust link.

It should be considered whether security should be enhanced by requiring users to enter a PIN number in order to pair the devices so that, for example, it is not possible for someone to steal your mobile phone and then use it with all of their Bluetooth enabled devices. Adding a requirement to enter a PIN number (where possible) would detract little from the usability advantages of the "Touch and Find" system and would add a security measure that is highly visible to the user giving increased peace of mind.

In summary, incorporating the "Touch and Find" system into devices will considerably enhance the users "Out of Box" experience, by improving the usability of the Bluetooth pairing mechanism to make it more intuitive and more secure. The "Touch and Find" system can be used with infrared, simple electrical contacts or an inductive loop as the physical medium with no software changes required. The inductive loop method may be extended to incorporate a universal battery charging method adding further value to the system. The "Touch and Find" system is ideal for use in a PAN Gateway and other Bluetooth devices that do not have a user interface.

8.2 SUMMARY

The initial concept for the MSc was to design and implement a PAN Gateway. But in the process of investigating the issues surrounding a PAN Gateway, in particular how Bluetooth pairing could be achieved simply without a user interface, a significant problem in the usability and security of Bluetooth pairing was discovered. The work changed focus to develop a system that would improve the usability of Bluetooth pairing in a way which was suitable for use in a PAN Gateway – a device with no user interface. A summary of the thesis is given below.

The thesis started in Chapters 1 and 2 by introducing what the PAN Gateway is and how it would be used and then moves on to discuss the usability in existing mobile devices. A comprehensive review of existing methods of text entry is given. In Chapter 3 the text described and evaluated the various technologies that could be used to provide the local connectivity of the PAN and investigate relevant coexistence and usability issues of Bluetooth. In Chapter 4 the requirements of the PAN Gateway were given, different user interfaces for the PAN Gateway were discussed and a novel concept for improving the usability of Bluetooth devices was introduced.

The thesis then shifted focus to develop and implement a system that improves the usability of Bluetooth pairing for use in Bluetooth devices, especially those such as the PAN Gateway that have a minimal user interface. In Chapter 5 the "Touch and Find"

system was introduced and the system requirements and proposed architecture were discussed. This was followed by an explanation of the design of the Pairing Link Protocol, which specifies the signal flow for the "Touch and Find" system. Chapter 5 finished with the implementation and testing of the main PLP software task.

In Chapter 6, the design, implementation and testing of the PLP Transport task was discussed with a detailed description of how the Windows serial port was used. Chapter 7 described the final link in the system, the different physical mediums that could be used. Three types of solution were introduced, a simple electrical contacts solution, infrared solutions and an inductive solution. Modifications of the software that were required in order to detect the "connected" state are documented. In Chapter 8 the evaluation, summary and conclusion of the work was given.

8.3 CONCLUSION

The research has investigated a wide variety of issues relating to the usability of the Personal Area Network Gateway. Initially the Man Machine Interface (MMI) for the PAN Gateway was considered including investigation into the existing text entry methods for mobile devices. Inefficient text entry methods are the source of one of the most significant usability problems in mobile devices. It was concluded that the best MMI for the PAN Gateway was minimal; a power button and an L.E.D.

The different technologies that could be used to provide the local connectivity for devices in the Personal Area Network were investigated and it was concluded that IrDA was unsuitable as it's restricted to line-of-sight operation. Of the possible RF technologies using the 2.4GHz ISM band, only Bluetooth was low power. Bluetooth also had the advantage of being designed for use by consumers as a cable replacement technology, making it the optimal technology for use in the PAN Gateway.

The PAN Gateway relies on Bluetooth technology to link with other devices to form the Personal Area Network and requires "pairing" of devices in order for them to communicate before the first link can be established. The Bluetooth pairing method in existing mobile devices was considered and determined to be time consuming, non-intuitive and unnecessarily complex, in addition to have questionable security measures when used in public places.

To improve the usability of Bluetooth devices and in particular the PAN Gateway, the "Touch and Find" system was developed. The "Touch and Find" system is based on the Pairing Link Protocol which specifies the required signal flow between devices to enable pairing and was designed to enable a robust link to be formed. The system uses a serial link over a variety of mediums to transfer the necessary information.

The "Touch and Find" system was implemented using simple electrical contacts and an inductive loop to create the serial link; Infrared solutions were also designed. In the simple electrical contact solution, the user simply has to touch the contacts from each of the two devices together to exchange the information.

The "Touch and Find" system could also be used for authorisation purposes in other wireless networks such as 802.11b. Indeed it could be used for a wide variety of authentication and authorisation procedures. In addition it could be used for synchronising devices or for other technologies in which two devices have to be setup to work together, for example a similar system could be used to tune a tv to a video.
The "Touch and Find" system has been shown to be a good concept. It was developed with the aim of improving the usability of Bluetooth pairing by making it intuitive and to also enable Bluetooth Pairing to be carried out securely in public places. This has been achieved; a robust, secure, intuitive method of Bluetooth pairing has been developed using a variety of different mediums (simple electrical contacts, Infrared and an Inductive loop). The addition of the "Touch and Find" system to Bluetooth enabled devices will add value to the devices as the user's "Out of Box" experience will be significantly enhanced, breaking down initial barriers that may prevent an individual from using Bluetooth technology.

8.4 Enhancements to the "Touch and Find" System

Software and Hardware

The PLP transport task could be made even more reliable by using Cyclic Redundancy Checks (CRC) on all signals transmitted across the hardware link. This would allow any errors in the data to be detected. The "random" number that is used as the Bluetooth link key should be made more "random" – this could be achieved by using the Bluetooth clock to generate the "random" number.

Further work should also be carried out on building and testing the contacts for use in the simple electrical contact solution. Particular attention should be paid to designing

contacts that are unlikely to be damaged and that are aesthetically pleasing so that potential users are not put off by the look of the contacts. The Infrared solutions should be built and tested to verify that the "Touch and Find" system works using an Infrared link. Finally, it would be very interesting and useful if a "two coil" full duplex inductive loop solution that was capable of being used to charge the mobile device's battery could be developed.

System

As far as the "Touch and Find" system is concerned, the next step is to develop an application interface to enable the results of the "Touch and Find" system to be used in a real application that actually creates the first Bluetooth link between two devices. The system should then be implemented on a TTPCom Bluetooth Evaluation Board before finally creating and testing a prototype and building it into a product!

The IrDA Infrared hardware solution could be improved (and the cost reduced) from a system point of view if an existing IrDA stack could be extended to support the "Touch and Find" process. This would mean that it would be very cheap to add the "Touch and Find" system to devices that have an IrDA port.

Finally, work should be done to develop an Inductive solution that was also capable of receiving sufficient power to charge the device's batteries. This would be a huge "value added" feature for mobile devices. It could enable a universal charger to be created, so that only one mobile device charger would need to be carried when travelling with the added advantage that the same interface could be used to quickly and simply pair Bluetooth devices.

8.5 CONCLUDING STATEMENT

The PAN Gateway is a revolutionary concept in mobile communication offering great advantages to users and operators. The optimal PAN Gateway user interface is minimal; it should consist of just a power button and a single L.E.D to indicate whether it is switched on. The PAN Gateway should consist of a Bluetooth modem to provide local connectivity, a GSM/GPRS modem to provide connection to mobile phone networks and some routing technology.

It was difficult to design a user friendly Bluetooth Pairing method for the PAN Gateway using existing technology, as there was no user interface. Indeed, Bluetooth pairing in many existing devices was found to be unintuitive and difficult. In addition, a Bluetooth SIG security white paper had advised that the existing Bluetooth Pairing procedure should not be used in public places.

To solve the usability and security problems encountered, a new concept, the "Touch and Find" system was developed for use in the PAN Gateway. The system uses a serial link to transfer the information required by devices in order to pair. The signal flow for the "Touch and Find" system is specified by the Pairing Link Protocol that was designed by the author. The Pairing Link Protocol was specifically designed to be robust. The "Touch and Find" system that was developed interfaces directly with the Bluetooth Device Manager. The "Touch and Find" system was shown to work using different physical mediums to link the two devices, including a simple electrical contact solution and an inductive loop solution. The “Touch and Find” system developed is quick and very robust.

The “Touch and Find” system developed by the author significantly improves the user’s “Out of Box” experience, by simplifying the Bluetooth Pairing procedure in addition to providing a secure means of Bluetooth pairing in public places. "Touch and Find" is a robust system that will add value to the devices it is used in and could be extended for use in other systems.

REFERENCES

[1] M. Gerla, R. Kapoor, M. Kazantzidis, and P. Johansson, "Ad-hoc Networking with Bluetooth," presented at WMI at Mobicom, Rome, Italy, 2001.

[2] J. Bray and C. Sturman, *Connect without Cables*, vol.: Prentice Hall PTR, 2001.

[3] M. Jakobsson and S. Wetzel, "Security Weaknesses in Bluetooth," Lucent Technologies, Bell Labs, Murray Hill, 2001.

[4] N. Rouhana and E. Horlait, "BWIG: Bluetooth Web Internet Gateway," presented at 7th IEEE ISCC, Toarmina, Italy, 2001.

[5] P. Ostergaard, "Evaluating Bluetooth for Telephony in the Enterprise.," presented at Bluetooth Congress 2001, Grimaldi Forum, Monaco., 2001.

[6] D. Sumption, "And the Winner is ... Mobile Phones or Handheld Computers", www.sumption.org/articles/marketing-MobileVsPDA.asp, 2001.

[7] J. Nielsen, "Mobile Phones: Europe's Next Minitel?"", Jakob Nielsen's Alertbox, http://www.useit.com/alertbox/20010107.html, 2001.

[8] J. Hawkins, "Treo", www.handspring.com, 2001.

[9] M. Megennis, "Mobile Phone Usability," 2001, http://infocentre.frontend.com/servlet/Infocentre?access=no&page=article&rows=5&id=92.

[10] D. Johnson, "Batteries Hold the Key to Mobility", It World, 2001.

[11] M. Wilner, "White Paper: High Speed Text Entry for Handhelds," Allnet devices, http://allnetdevices.com/developer/white/2001/02/13/high-speed_text.html, Feb 13th 2001 2001.

[12] A. McClard and P. Somers, "Unleashed, Web Tablet Integration into the Home," *proc. of CHI'2000: Human Factors in Computing Systems. 2000. The Hague, The Netherlands.*, pp. p1 - 8, 2000,

[13] S. Zhai, M. Hunter, and B. Smith, "The Metropolis Keyboard - An Exploration of Quantitative Techniques for Virtual Keyboard Design," presented at ACM Symposium on User Interface Software and Technology (UIST 2000), 2000.

[14] E. Baig, "Demo Show Rounds up the Latest in Wireless Wonders", Usa Today, www.usatoday.com/life/cyber/ccarch/2001/09/12/baig.htm, 2001.

[15] J. Nielsen, "Mobile Devices will soon be Useful", Jakob Nielson's Alertbox, http://www.useit.com/alertbox/20010916.html, 2001.

[16] N. F. Ayan, B. Karagol-Ayan, D. Kuehurt, and A. Thakkar, "SHORE 2001 : Which is Faster and More Accurate on a Handheld : Graffiti or Keyboard Tapping?," 2001,

[17] M. J. LaLomia, "User Acceptance of Handwritten Recognition Technology," *proc. of CHI'94*, pp. p107, 1994,

[18] I. S. MacKenzie and S. X. Zhang, "The design and evaluation of a high-performance soft keyboard," *proc. of CHI'99: ACM Conference on Human Factor in Computing Systems*, pp. p25 - 31, 1999,

[19] T. Bellman and I. S. MacKenzie, "A Probabilistic Character Layout for Mobile Text Entry," *Proc. of Graphics Interface '98, Toronto: Canadian Information Processing Society*, pp. pp 168-176, 1998,

[20] "A Survey of Alternate Text Entry Methods," Eatoni Ergonomics, http://www.eatoni.com/research/alternates.pdf, 2000.

[21] J. Nielson, "New Devices Augur Decent Mobile Experience", Jakob Nielsen's Alertbox, http://www.useit.com/alertbox/20000917.htm, 2000.

[22] M. Silfverberg, I. Mackenzie, and P. Korhonen, "Predicting Text Entry Speeds in Mobile Phones," presented at ACM Conference on Human Factors in Computing Systems. CHI'2000, New York, 2000.

[23] Z. Friedman, S. Mukherji, G. K. Roeum, and R. Ruchir, "SHORE 2001: Data Input into Mobile Phones: T9 or Keypad?," 2001,

[24] S. Weiss, *Handheld Usability*, vol.: John Wiley and Sons, 2002.

[25] C.-M. Karat, C. Halverson, D. Horn, and J. Karat, "Patterns of Entry and Corrections in Large Vocabulary Speech Recognition Systems.," 1999,

[26] E. Batista, "Speaking of Voice Recognition", www.wired.com/news/print/0,1294,47545,00, 2001.

[27] D. Suvak, "IrDA and Bluetooth: A Complementary Comparison," Extended Systems, Inc, http://www.irda.org/design/ESIIrDA_Bluetoothpaper.doc, 2000.

[28] G. Yeadon, "What are the coming trends in the wireless world and what are their implications in the home, the office, and elsewhere?," Richmond, http://www.student.richmond.edu/2001/gyeadon/portfolio/projects/itsite/FrameSet2.html, 2001.
[29] M. Dempsey, "The Physiological effects of 2.4GHz Frequency Hopping Radios," Hewlett Packard Company, 2001.
[30] P. Mars, "Digital Communication Lecture Notes; Spread Spectrum Modulation," Durham University, 2001.
[31] N. J. Muller, *Bluetooth Demystified*, vol.: McGraw Hill, 2001.
[32] A. Kansal and U. B. Desai, "Mobility Support for Bluetooth Access," Department of Electrical Engineering, Indian Institute of Technology Bombay, Mumbai-400076, India, 2001.
[33] D. Suvak, "Supporting Bluetooth Multipoint Networks," presented at Bluetooth Congress, Grimaldi Forum, Monaco, 2001.
[34] "Bluetooth and Successor 802.15.3," Wireless World Forum, http://www.wirelessworldforum.com/printout.php?item=10566&s=0&n=10, 9th October 2001.
[35] E. Tay, "Comparing Infrared and Bluetooth Short-Range Solutions," *Microwaves and RF*, 2001,
[36] C. d. M. Cordeiro and D. P. Agrawal, "Mitigating the Effects of Intermittent Interference on Bluetooth Ad Hoc Networks," Center for Distributed and Mobile Computing, ECECS, Cincinnati, 2001.
[37] "Interference and Coexistence," Code Blue Communications, http://www.codebluecommunications.com/documents/coexistence%20whitepaper.pdf, 2001.
[38] J. Zyren, "Extension of Bluetooth and 802.11 Direct Sequence Interference Model," *IEEE802.11-98/378*, 1998,
[39] J. Zyren, "Reliability of IEEE 802.11 DSSS WLANs in a high density Bluetooth Environment," *Intersil Corporation - Prism Products*, 1999,
[40] G. Ennis, "Impact of Bluetooth on 802.11 Direct Sequence Wireless LAN's," *IEEE802.11-98/319a*, 1998,
[41] J. Lansford, R. Nevo, and B. Monello, "Wi-Fi (802.11b) and Bluetooth Simultaneous Operation: Characterising the Problem," Mobilian Corporation, http://www.mobilian.com/documents/MobilianWhitepaper.pdf, 2000.
[42] "Wi-Fi (IEEE 802.11b) and Bluetooth - Coexistence Issues and Solutions for the 2.4GHz Band," Texas Instruments, 2001.
[43] "Interference Immunity of 2.4GHz Wireless LANs," HomeRf Working Group, www.homerf.org, 2001.
[44] J. Hartsen and S. Zurbes, "Bluetooth voice and data performance in 802.11 DS WLAN environment", http://www.wirelessethernet.org/downloads/BT_inf802_June_8.pdf, 1999.
[45] "Study Finds No Bluetooth-802.11 Interference," Allnet Devices, http://www.80211-planet.com/news/article/0,,1481_937781,00.html, 14th December 2001.
[46] "Adaptive Frequency Hopping : Good Enough?," Mobilian Corporation, www.mobilian.com/images/AFH-final.pdf, 2002.
[47] M. Peretz, "Companies to Show Bluetooth, WLAN Coexistence", 802-11 planet, http://www.80211-planet.com/news/article/0,,1481_937781,00.html, 2001.
[48] M. Peretz, "Detente in the Airwaves: 802.11 and Bluetooth Together," 2001, http://www.80211-planet.com/news/article/0,,1781_936271,00.html.
[49] C. Gehrmann, "Bluetooth Security White Paper," Bluetooth SIG, 2002.
[50] S. Ohr, "Bluetooth Projected to Emerge in Force Next Year", EE Times, www.eetimes.com/story/OEG20011214S0076, 2001.
[51] "The IXI Platform," IXI Mobile Inc., http://www.ixi.com/PDF/IXI_Brochure.pdf, 2002.
[52] "Specification of the Bluetooth System Core, version 1.1," Bluetooth SIG, 2001.
[53] J. Haine, "Two-wire duplex data transmission," P. Regan, Ed., 2002.
[54] "Technical Data IR 3/16 Encode/Decode IC," Agilent Technologies, www.semiconductor.agilent.com, 2000.
[55] "Infrared IrDa Compliant Transceiver Technical Data," Hewlett Packard Company, 1996.

APPENDIX 1

Nassi Schneiderman diagram of plpbuProcessRxData 2

PLPBU_PROCESS_RX_DATA		
case PLPTX_BUS_RX_PACKET_TYPE	case PLPTX_BUS_RX_SIGNAL	default:
PLPTX_BUS_RX_PACKET_TYPE	PLPTX_BUS_RX_SIGNAL	DevFail (" incorrect state")

PLPTX_BUS_RX_PACKET_TYPE			
INFO_TYPE	ACK_TYPE	START_TYPE	default:
set number of bytes to be read = 272	set number of bytes to be read = 2	set number of bytes to be read = 2	DevFail (" incorrect type")
reset receive Buffer pointer	reset receive Buffer pointer	reset receive Buffer pointer	

PLPTX_BUS_RX_SIGNAL		
START_SIGNAL	START_SIGNAL2	PLPTX_BUS_OUT_INFO
create PLPTX_START_SEQUENCE2_REQ	create PLPTX_START_SEQUENCE_CNF	create PLPTX_IN_INFO_IND
fill and send signal to PLPTX task	fill and send signal to PLPTX task	fill and send to PLP task PLPTX_IN_INFO_IND

PLPTX_BUS_RX_SIGNAL	
PLPTX_BUS_ACK	default:
create PLPTX_OUT_INFO_CNF	DevFail (" incorrect signal")
fill and send to PLP task PLPTX_OUT_INFO_CNF	

APPENDIX 2

Nassi Schneiderman diagrams of plpbuProcessRxData 3

PLPBU_PROCESS_RX_DATA		
case PLPTX_BUS_START	case PLPTX_BUS_RX_REMOTE_DEVICE_ID	case PLPTX_BUS_RX_PACKET_TYPE
PLPTX_BUS_START	PLPTX_BUS_RX_REMOTE_DEVICE_ID	PLPTX_BUS_RX_PACKET_TYPE

PLPBU_PROCESS_RX_DATA	
case PLPTX_BUS_RX_SIGNAL	default:
PLPTX_BUS_RX_SIGNAL	DevFail (" incorrect type")

PLPTX_BUS_START		
True — rxByte = PRE_AMBLE_BYTE — False		
startByteCounter++;		discard Byte
True — startByteCounter = 3? — False		PLPTX_BUS_START
PLPTX_BUS_RX_REMOTE_DEVICE_ID	PLPTX_BUS_START	

PLPTX_BUS_RX_REMOTE_DEVICE_ID	
True — if remoteDeviceId = localDeviceId — False	
discard	PLPTX_BUS_RX_PACKET_TYPE
PLPTX_BUS_START	

PLPTX_BUS_RX_PACKET_TYPE			
case INFO_TYPE	case ACK_TYPE	START_TYPE	default:
set number of bytes to be read	set number of bytes to be read	set number of bytes to be read	discard
reset receive Buffer pointer	reset receive Buffer pointer	reset receive Buffer pointer	

PLPTX_BUS_RX_SIGNAL		
START_SIGNAL	START_SIGNAL2	case PLPTX_BUS_OUT_INFO
create PLPTX_START_SEQUENCE2_REQ	create PLPTX_START_SEQUENCE_CNF	create PLPTX_IN_INFO_IND
fill and send signal to PLPTX task	fill and send signal to PLPTX task	fill and send to PLP task PLPTX_IN_INFO_IND
		reset to receive Start
		PLPTX_BUS_START

PLPTX_BUS_RX_SIGNAL	
case PLPTX_BUS_ACK	default:
create PLPTX_OUT_INFO_CNF	discard
fill and send to PLP task PLPTX_OUT_INFO_CNF	
reset to receive Start	
PLPTX_BUS_START	

APPENDIX 3 CODE FOR THE TOUCH AND FIND SYSTEM

```
/*****************************************************************************
 * * $Workfile:   plp_fnc.c  $
 * $Revision:
 *    $Date:
 *****************************************************************************
 * Designed by    : PKR
 * Coded by       :
 * Tested by      : PKR
*****************************************************************************/
#define MODULE_NAME "PLP_FNC"

/*****************************************************************************
 * Include Files
 *****************************************************************************/

#if defined (HPDEFINE)
# if !defined (HPDEFINE_H)
#  include "hpdefine.h"
# endif
#endif

#if !defined (STRING_H)
# include "string.h"
#endif

#if !defined (KERNEL_H)
# include "kernel.h"
#endif

#if !defined (PLP_SIG_H)
#include "plp_sig.h"
#endif

#if !defined (PLPTX_SIG_H)
#include "plptx_sig.h"
#endif

#if !defined (PLP_FNC_H)
#include "plp_fnc.h"
#endif

#if !defined (PLP_TYP_H)
#include "plp_typ.h"
#endif

#if !defined (PLPMN_FNC_H)
#include "plpmn_fnc.h"
#endif

#if !defined (PLPTXMN_FNC_H)
#include "plptxmn_fnc.h"
#endif
```



```
#if !defined (PLPTXBU_TYP_H)
#include "plptxbu_typ.h"
#endif

#if defined (PLP_TRACE_OUTPUT)
# include "pssignal.h"
# include "emmi_sig.h"
# include "stdio.h"
#endif


/*****************************************************************************
 * Manifest Constants
 *****************************************************************************/

/*****************************************************************************
 * Types
 *****************************************************************************/
typedef enum PlpStateTag
{
 IDLE,
 ACTIVE,
 GOT_KEY,
 WAIT_FOR_KEY
}PlpState;

 typedef struct PlpContextTag

 Boolean          signalHandled;
 Boolean          plpStateTimerRunning;
 Boolean          plpSendIntervalTimerRunning;
 PlpState         plpState;
 PlpState         oldState;
 PlpDeviceInfo    plpLocalDeviceInfo;
 PlpDeviceInfo    plpRemoteDeviceInfo;
 Int32           plpSendIntervalCounter;
 Int32           plpStateCounter;
 TaskId           taskId;
 Boolean          sentFinishReq;
 Boolean          receiveFinishInd;
}PlpContext;

/*****************************************************************************
 * General Variables
 *****************************************************************************/

PlpContext        plpContext;
#if defined (PLP_TRACE_OUTPUT)
extern char traceString[ MAX_TEST_FILE_OUT_STRING ];
#endif

/*****************************************************************************
 * Signal Variables
 *****************************************************************************/

/*****************************************************************************
 * Timer Variables
```

```
 ***************************************************************************/
KiTimer         plpSendIntervalTimer;
KiTimer         plpStateTimer;

/***************************************************************************
 * Macros
 ***************************************************************************/

/***************************************************************************
 * Functional Prototypes
 ***************************************************************************/

void plpInitTimers (void);
void plpStartSendIntervalTimer (void);
void plpStopSendIntervalTimer (void);
void plpStartStateTimer (void);
void plpStopStateTimer (void);
void plpStateTimerRunning (Boolean);
void plpSendIntervalTimerRunning (Boolean);
void plpIdleState (SignalBuffer *);
void plpActiveState (SignalBuffer *);
void plpWaitForKeyState (SignalBuffer *);
void plpGotKeyState (SignalBuffer *);
void plpAllInfoState (SignalBuffer *);
Int8 plpRandom(Int8);
void plpSendInfo (void);
void plpInInfo (SignalBuffer *);
void plpLinkInfo (void);
void plptxInInfoRsp (void);
void plptxOutFinishReq(void);


/***************************************************************************
 *     Global Functions
 ***************************************************************************/

/***************************************************************************
* Function:    plpSwitch
*
* Description:
***************************************************************************/

void plpSwitch   (SignalBuffer * signalBuffer_p)
{
 plpSignalHandled (TRUE);

 switch ( *signalBuffer_p->type)
 {
  case SIG_TIMER_EXPIRY:

    /* if its the sendInterval timer the deal with it in the state switch (not here)*/
    if (signalBuffer_p->sig->kiTimerExpiry.timerId == plpSendIntervalTimer.timerId)
    {
     plpSignalHandled (FALSE);

     /* call state switch */
     plpStateSwitch (signalBuffer_p);
    }
```



```
    else
    {
     if (signalBuffer_p->sig->kiTimerExpiry.timerId == plpStateTimer.timerId)
     {
      if (plpContext.plpStateTimerRunning == TRUE)
      {
#if defined (PLP_TRACE_OUTPUT)
        sprintf(traceString,"PLP: In state too long - going to IDLE state",*signalBuffer_p->type);
        plpTraceOutput(traceString);
#endif

#if defined (PLP_TRACE_OUTPUT)
        sprintf(traceString,"PLP: SendInterval TIMER_EXPIRY occurred");
        plpTraceOutput(traceString);
#endif

        plpContext.plpState = IDLE;
       }
       else
       {
        /* Do Nothing as the timer has now been stopped - plpContext.plpStateTimerRunning = FALSE*/

#if defined (PLP_TRACE_OUTPUT)
        sprintf(traceString,"PLP: State TIMER_EXPIRY ignored as timer stopped");
        plpTraceOutput(traceString);
#endif
       }
      }
      else
      {
       /* signal used elsewhere - let it through */
       plpSignalHandled (FALSE);
       plpStateSwitch (signalBuffer_p);
      }
     }
     break;


   default:
    plpStateSwitch (signalBuffer_p);
    break;
 }

}
/***************************************************************************
* Function:    plpStateSwitch
*
* Description:  Main switch state machine
***************************************************************************/

void plpStateSwitch (SignalBuffer * signalBuffer_p)
{
 switch (plpContext.plpState)
  {
   case IDLE:
    plpIdleState(signalBuffer_p);
    break;
```

```
    case ACTIVE:
      plpActiveState (signalBuffer_p);
      break;

    case GOT_KEY:
      plpGotKeyState (signalBuffer_p);
      break;

    case WAIT_FOR_KEY:
      plpWaitForKeyState(signalBuffer_p);
      break;

    default:
      DevFail ("Unknown State");
      break;
   }
}

/*****************************************************************************
 * Function:    plpIdleState
 *
 * Description:
 *****************************************************************************/
void plpIdleState (SignalBuffer * signalBuffer_p)
{
 SignalBuffer signalToSend = kiNullBuffer ;
 Int8 index ;

#if defined (PLP_TRACE_OUTPUT)
    if (plpContext.plpState != plpContext.oldState)
    {
     sprintf(traceString,"PLP: PLP in IDLE STATE");
     plpTraceOutput(traceString);
     plpContext.oldState = plpContext.plpState;
    }
#endif

 switch (*(signalBuffer_p->type))
 {
  case SIG_DMSH_REGISTER_APPLICATION_CNF:

    DevAssert (signalBuffer_p->sig->dmshRegisterApplicationCnf.comStatus == COMMAND_OK);

    if (signalBuffer_p->sig->dmshRegisterApplicationCnf.comStatus != COMMAND_OK)
    {
     plpSignalHandled (FALSE);
    }

    /* ask Device manager for local info - create signal */
    KiCreateSignal  (SIG_DMSH_READ_LOCAL_INFO_REQ,
              sizeof (DmshReadLocalInfoReq),
              &signalToSend);

    /* fill values into signal */
    signalToSend.sig->dmshReadLocalInfoReq.taskId = PLP_TASK_ID;

    /* send signal */
    KiSendSignal (DM_TASK_ID,&signalToSend);
```



```
#if defined (PLP_TRACE_OUTPUT)
    sprintf(traceString,"PLP: Expecting DMSH_READ_LOCAL_INFO_IND from device Manager - DM running?");
    plpTraceOutput(traceString);
#endif
     break;


   case SIG_PLPTX_OUT_INFO_CNF:
#if defined (PLP_TRACE_OUTPUT)
    sprintf(traceString,"PLP: received PLPTX_OUT_INFO_CNF in wrong state - ignore it");
    plpTraceOutput(traceString);
#endif
   break;

   case SIG_PLPTX_IN_FINISH_IND:
#if defined (PLP_TRACE_OUTPUT)
    sprintf(traceString,"PLP: received PLPTX_IN_FINISH_IND in wrong state - ignore it");
    plpTraceOutput(traceString);
#endif
   break;

   case SIG_PLPTX_START_SEQUENCE_CNF:
#if defined (PLP_TRACE_OUTPUT)
    sprintf(traceString,"PLP: received start sequence cnf, now connected");
    plpTraceOutput(traceString);
#endif

    plptxContext.busConnected = TRUE;

    /* stores TaskId from plpStartScanReq as plpContext.taskId for use in plpStartScanCnf*/
    memcpy (&plpContext.taskId,
         &signalBuffer_p->sig->plptxStartSequenceCnf.myTaskId,
         sizeof (TaskId));

    /* ask Device manager for local info - create signal */
    KiCreateSignal  (SIG_DMSH_READ_LOCAL_INFO_REQ,
              sizeof (DmshReadLocalInfoReq),
              &signalToSend);

    /* fill values into signal */
    signalToSend.sig->dmshReadLocalInfoReq.taskId = PLP_TASK_ID;

    /* send signal */
    KiSendSignal (DM_TASK_ID,&signalToSend);

#if defined (PLP_TRACE_OUTPUT)
    sprintf(traceString,"PLP: Expecting DMSH_READ_LOCAL_INFO_IND from device Manager - DM running?");
    plpTraceOutput(traceString);
#endif
   break;

  case SIG_DMSH_READ_LOCAL_INFO_CNF:
    if (signalBuffer_p->sig->dmshReadLocalInfoCnf.comStatus != COMMAND_OK)
    {
     DevFail ("Read Local Info Request Failed");
    }
    break;
```

```
  case SIG_DMSH_READ_LOCAL_INFO_IND:
    if (plptxContext.busConnected ==TRUE)
    {
     /* stores BtBdAddr as plpLocalDeviceInfo.plpBtBdAddr */
     memcpy (&plpContext.plpLocalDeviceInfo.plpBtBdAddr,
          &signalBuffer_p->sig->dmshReadLocalInfoInd.btBdAddr,
          sizeof (BtBdAddr));

     /* stores FriendlyName as PlpLocalDeviceInfo.plpFriendlyName */
     memcpy (&plpContext.plpLocalDeviceInfo.plpFriendlyName,
          &signalBuffer_p->sig->dmshReadLocalInfoInd.friendlyName,
          HCIE_07_NAME_SIZE);

     /* Generate and store a random link key */
     for (index=0 ; index < BT_ENCRYPTION_KEY_SIZE ; index++)
     {
      plpContext.plpLocalDeviceInfo.plpLinkKey[index] = plpRandom(index);
     }

     /* Start SendIntervalTimer */
     plpStartSendIntervalTimer ();

     /* start StateTimer */
     plpStartStateTimer ();
#if defined (PLP_TRACE_OUTPUT)
     sprintf(traceString,"PLP: PLP changing from IDLE to ACTIVE");
     plpTraceOutput(traceString);
#endif

     plpContext.plpState = ACTIVE;
    }
    else
    {
     /* stores BtBdAddr as plpLocalDeviceInfo.plpBtBdAddr */
      memcpy (&plptxContext.localDeviceId,
           &signalBuffer_p->sig->dmshReadLocalInfoInd.btBdAddr,
           sizeof (BtBdAddr));

      /* activate start sequence (part 1 of 2)*/
     KiCreateSignal      (SIG_PLPTX_START_SEQUENCE_REQ,
                sizeof (PlptxStartSequenceReq),
                &signalToSend);

     signalToSend.sig->plptxStartSequenceReq.myTaskId = PLP_TASK_ID;
     signalToSend.sig->plptxStartSequenceReq.localBtBdAddr = plptxContext.localDeviceId;

     KiSendSignal  (PLPTX_TASK_ID, &signalToSend);
    }
    break;

  case SIG_PLPTX_IN_INFO_IND:
    /* not ready to receive this signal. Ignore it for the time being */

#if defined (PLP_TRACE_OUTPUT)
    sprintf(traceString,"PLP: received PLPTX_IN_INFO_IND in wrong state- ignore it!");
    plpTraceOutput(traceString);
#endif
```



```
    /* Mark signal as handled */
    plpSignalHandled (TRUE);
    break;

  case SIG_TIMER_EXPIRY:  /* shouldn't get this here - but just in case */
    /* If it's the SendIntervalTimer that has expired */
    if (signalBuffer_p->sig->kiTimerExpiry.timerId == plpSendIntervalTimer.timerId)
    {
     if (plpContext.plpSendIntervalTimerRunning == TRUE)
     {
#if defined (PLP_TRACE_OUTPUT)
       sprintf(traceString,"PLP: Incorrect - SendInterval TIMER_EXPIRY occurred in IDLE state");
       plpTraceOutput(traceString);
#endif
     }
     else
     {
      /* do nothing as timer has already been stopped - plpContext.plpSendIntervalTimerRunning = FALSE */

#if defined (PLP_TRACE_OUTPUT)
       sprintf(traceString,"PLP: SendInterval TIMER_EXPIRY ignored as timer stopped");
       plpTraceOutput(traceString);
#endif
     }
    }
    else
    {
     /* signal for use elsewhere - let it through */
     plpSignalHandled (FALSE);
    }
   break;


  default:
    plpSignalHandled (FALSE);
    break;
  }
}


/*****************************************************************************
* Function:    plpActiveState
*
* Description:
*****************************************************************************/

void plpActiveState (SignalBuffer * signalBuffer_p)
{
#if defined (PLP_TRACE_OUTPUT)
 if (plpContext.plpState != plpContext.oldState)
 {
  sprintf(traceString,"PLP: PLP in ACTIVE State");
  plpTraceOutput(traceString);
  plpContext.oldState = plpContext.plpState;
 }
#endif
```

```
 switch (*(signalBuffer_p->type))
 {
  case SIG_TIMER_EXPIRY:
   /* if its the state timer it has been dealt with earlier in plpSwitch() */

   /* If it's the SendIntervalTimer that has expired */
   if (signalBuffer_p->sig->kiTimerExpiry.timerId == plpSendIntervalTimer.timerId)
   {
    if (plpContext.plpSendIntervalTimerRunning == TRUE)
    {
#if defined (PLP_TRACE_OUTPUT)
      sprintf(traceString,"PLP: SendInterval TIMER_EXPIRY occurred in ACTIVE state");
      plpTraceOutput(traceString);
#endif

      plpSendInfo ();  /* sends PLP_OUT_INFO_REQ */

      /* start send interval timer */
      plpStartSendIntervalTimer ();
     }
     else
     {
      /* do nothing as the timer has already been stopped - plpContext.plpSendIntervalTimerRunning = FALSE */

#if defined (PLP_TRACE_OUTPUT)
      sprintf(traceString,"PLP: SendInterval TIMER_EXPIRY ignored as timer stopped");
      plpTraceOutput(traceString);
#endif
     }
    }

    else
    {
     /* signal for use elsewhere let it through */
     plpSignalHandled(FALSE);
    }
    break;


   case SIG_PLPTX_IN_INFO_IND:
    plpSignalHandled(TRUE);

    if (signalBuffer_p->sig->plptxInInfoInd.plpStatus == PLP_COMMAND_OK)
    {
     plpInInfo (signalBuffer_p);  /* uses function to store plpInInfoInd */

     plptxInInfoRsp ();     /* function to send plptxInInfoRsp */

     /* Start StateTimer */
     plpStartStateTimer ();

     /* start sendInterval Timer */
     plpStartSendIntervalTimer ();

#if defined (PLP_TRACE_OUTPUT)
     sprintf(traceString,"PLP: PLP State changing from ACTIVE to GOT_KEY");
     plpTraceOutput(traceString);
#endif
```



```
     plpContext.plpState = GOT_KEY;
    }

    else
    {
     /* stay in active state - do nothing */
     DevFail ("Incoming information != PLP_COMMAND_OK");
    }
    break;

   case SIG_PLPTX_OUT_INFO_CNF:

    if (signalBuffer_p->sig->plptxOutInfoCnf.plpStatus == PLP_COMMAND_OK)
    {
     /* stop sendInterval timer */
     plpStopSendIntervalTimer ();

     plpStartStateTimer ();

#if defined (PLP_TRACE_OUTPUT)
     sprintf(traceString,"PLP: PLP State changing from ACTIVE to WAIT_FOR_KEY");
     plpTraceOutput(traceString);
#endif

     plpContext.plpState = WAIT_FOR_KEY;
    }
    else
    {
     DevFail ("Outgoing info not received");
    }
    break;

   default:
    plpSignalHandled (FALSE);
    break;
    }
}

/*****************************************************************************
* Function:   plpWaitForKeyState
*
* Description:
*****************************************************************************/

void plpWaitForKeyState (SignalBuffer * signalBuffer_p)
{
#if defined (PLP_TRACE_OUTPUT)
  if (plpContext.plpState != plpContext.oldState)
  {
   sprintf(traceString,"PLP: PLP in WAIT_FOR_KEY State");
   plpTraceOutput(traceString);
   plpContext.oldState = plpContext.plpState;
  }
#endif

 switch (*(signalBuffer_p->type))
 {
```

```
  case SIG_PLPTX_IN_INFO_IND:
   if (signalBuffer_p->sig->plptxInInfoInd.plpStatus == PLP_COMMAND_OK)
   {
    plpInInfo (signalBuffer_p);  /* uses function to store plpInInfoInd and to create and send plpInInfoRsp */

    plptxInInfoRsp (); /* send Rsp to pltx task */

    plpLinkInfo (); /* function to create and send plpLinkInfoInd*/

    sprintf(traceString,"PLP: This device has all info");
    plpTraceOutput(traceString);

    /* send the plptxOutFinishReq signal */
    plptxOutFinishReq ();

    plpContext.sentFinishReq = TRUE;

    if (plpContext.receiveFinishInd == TRUE)
    {
     /* Stop StateTimer */
     plpStopStateTimer ();

     /* finished go back to IDLE state */
     plpContext.plpState = IDLE;

#if defined (PLP_TRACE_OUTPUT)
      sprintf(traceString,"PLP: PLP Finished.  State changing from WAIT_FOR_KEY to IDLE");
      plpTraceOutput(traceString);
#endif
    }
    else
    {
#if defined (DEBUG_PLPTX)
      sprintf(traceString,"PLP: waiting for plptxInFinishInd");
      plpTraceOutput(traceString);
#endif
    }
   }

   else
   {
    /* do nothing - stay in WAIT_FOR_KEY_STATE */
   }

   break;


  case SIG_TIMER_EXPIRY:
  /* If it's the SendIntervalTimer that has expired */
   if (signalBuffer_p->sig->kiTimerExpiry.timerId == plpSendIntervalTimer.timerId)
   {
    if (plpContext.plpSendIntervalTimerRunning == TRUE)
    {
#if defined (PLP_TRACE_OUTPUT)
      sprintf(traceString,"PLP: Incorrect - SendInterval TIMER_EXPIRY occurred in WAIT_FOR_KEY state");
      plpTraceOutput(traceString);
#endif
    }
```



```
    else
    {
     /* do nothing as timer has already been stopped - plpContext.plpSendIntervalTimerRunning = FALSE */

#if defined (PLP_TRACE_OUTPUT)
      sprintf(traceString,"PLP: SendInterval TIMER_EXPIRY ignored as timer stopped");
      plpTraceOutput(traceString);
#endif
    }
   }
   else
   {
    /* signal for use elsewhere - let it through */
    plpSignalHandled (FALSE);
   }
  break;

  case SIG_PLPTX_OUT_INFO_CNF:

   /* This signal has already been received so stay in this state */
   break;

  case SIG_PLPTX_IN_FINISH_IND:
   /* other device has all the info it needs */
   sprintf(traceString,"PLP: Other device has finished");
   plpTraceOutput(traceString);

   plpContext.receiveFinishInd = TRUE;

   if (plpContext.sentFinishReq == TRUE)
   {
    /* Stop StateTimer */
    plpStopStateTimer ();

    /* finished go back to IDLE state */
    plpContext.plpState = IDLE;

#if defined (PLP_TRACE_OUTPUT)
     sprintf(traceString,"PLP: PLP Finished.  State changing from WAIT_FOR_KEY to IDLE");
     plpTraceOutput(traceString);
#endif
   }
   else
   {
#if defined (DEBUG_PLPTX)
     sprintf(traceString,"PLP: received finishInd, but I'm not ready to finish- ignoring it");
     plpTraceOutput(traceString);
#endif
   }
   break;

  default:
   plpSignalHandled (FALSE);
   break;
 }
}

/*****************************************************************************
```

```
 * Function:    plpGotKeyState
 *
 * Description:
 ******************************************************************************/
void plpGotKeyState (SignalBuffer * signalBuffer_p)
{
 SignalBuffer signalToSend = kiNullBuffer ;

#if defined (PLP_TRACE_OUTPUT)
   if (plpContext.plpState != plpContext.oldState)
   {
    sprintf(traceString,"PLP: PLP in GOT_KEY State");
    plpTraceOutput(traceString);
    plpContext.oldState = plpContext.plpState;
   }
#endif

  switch (*(signalBuffer_p->type))
  {
   case SIG_TIMER_EXPIRY:

    /* If it's the SendIntervalTimer that has expired */
    if (signalBuffer_p->sig->kiTimerExpiry.timerId == plpSendIntervalTimer.timerId)
    {
     if (plpContext.plpSendIntervalTimerRunning == TRUE)
     {
#if defined (PLP_TRACE_OUTPUT)
       sprintf(traceString,"PLP: SendInterval TIMER_EXPIRY occurred in GOT_KEY state");
       plpTraceOutput(traceString);
#endif
       plpSendInfo ();  /* sends PLP_OUT_INFO_REQ */

       /* start send interval timer */
       plpStartSendIntervalTimer ();
     }
     else
     {
      /* do nothing as timer has already been stopped - plpContext.plpSendIntervalTimerRunning = FALSE */

#if defined (PLP_TRACE_OUTPUT)
       sprintf(traceString,"PLP: SendInterval TIMER_EXPIRY ignored as timer stopped");
       plpTraceOutput(traceString);
#endif
     }
    }
    else
    {
     /* signal for use elsewhere - let it through */
     plpSignalHandled (FALSE);
    }
    break;

   case SIG_PLPTX_IN_INFO_IND:
    /* already know this but send a rsp anyway */

    plptxInInfoRsp (); /* send Rsp to PLPTX */

    break;
```



```
   case SIG_PLPTX_OUT_INFO_CNF:

    if (signalBuffer_p->sig->plptxOutInfoCnf.plpStatus == PLP_COMMAND_OK)
    {
     /* stop sendIntervalTimer */
     plpStopSendIntervalTimer ();

     /* stop state timer */
     plpStopStateTimer ();

     /* all info now received - create signal to send to application */
     plpLinkInfo();

     sprintf(traceString,"PLP: This device has all info");
     plpTraceOutput(traceString);

     /* send the plptxOutFinishReq signal */
     plptxOutFinishReq ();
     plpContext.sentFinishReq = TRUE;

     if (plpContext.receiveFinishInd == TRUE)
     {
      /* finished go back to IDLE state */
      plpContext.plpState = IDLE;

#if defined (PLP_TRACE_OUTPUT)
       sprintf(traceString,"PLP: PLP Finished.  State changing from GOT_KEY to IDLE");
       plpTraceOutput(traceString);
#endif
     }
     else
     {
#if defined (DEBUG_PLPTX)
       sprintf(traceString,"PLP: waiting for plptxInFinishInd");
       plpTraceOutput(traceString);
#endif
     }
    }
    else
    {
     /* do nothing - stay in GOT_KEY_STATE */
    }

    break;


   case SIG_PLPTX_IN_FINISH_IND:
    /* other device has all the info it needs */
    sprintf(traceString,"PLP: Other device has finished");
    plpTraceOutput(traceString);

    plpContext.receiveFinishInd = TRUE;

    if (plpContext.sentFinishReq == TRUE)
    {
     /* finished go back to IDLE state */
```

```
    plpContext.plpState = IDLE;

#if defined (PLP_TRACE_OUTPUT)
    sprintf(traceString,"PLP: PLP Finished.  State changing from GOT_KEY to IDLE");
    plpTraceOutput(traceString);
#endif
   }
   else
   {
#if defined (DEBUG_PLPTX)
    sprintf(traceString,"PLP: received finishInd, but I'm not ready to finish- ignoring it");
    plpTraceOutput(traceString);
#endif
   }
   break;

  default:
   plpSignalHandled (FALSE);
   break;

 }

}

/*****************************************************************************
 * Function:    plpInit
 *
 * Description: Plp Initialisation routine - called when SIG_INITIALISE
 *              is received
 *****************************************************************************/
void plpInit(void)
{
 SignalBuffer signalToSend = kiNullBuffer ;
 DmshRegisterApplicationReq * dmshRegisterApplicationReq_p;

 plpContext.sentFinishReq = FALSE;
 plpContext.receiveFinishInd = FALSE;

 plptxContext.busConnected = FALSE;

 plpContext.plpState = IDLE;

#if defined (PLP_TRACE_OUTPUT)
  plpContext.oldState = IDLE;
  sprintf(traceString,"PLP: IDLE State");
  plpTraceOutput(traceString);
#endif

 KiCreateSignal          (SIG_DMSH_REGISTER_APPLICATION_REQ,
            sizeof (DmshRegisterApplicationReq),
            &signalToSend);

 dmshRegisterApplicationReq_p = &signalToSend.sig->dmshRegisterApplicationReq;

 memset (dmshRegisterApplicationReq_p,0,sizeof(DmshRegisterApplicationReq));

 dmshRegisterApplicationReq_p->taskId = PLP_TASK_ID;
 dmshRegisterApplicationReq_p->registerAsApplication= TRUE;
```



```
 plpInitTimers ();

 KiSendSignal (DM_TASK_ID,&signalToSend);
}

/*****************************************************************************
 * Function:    plpRandom
 *
 * Description: returns a "random" Int8
 *****************************************************************************/
Int8 plpRandom(Int8 random)
{
 Int8 rand;

 rand = (Int8) ((random+2) * ((Int8) plpContext.plpLocalDeviceInfo.plpBtBdAddr.lap +
      (Int8) (plpContext.plpLocalDeviceInfo.plpBtBdAddr.uap << random )+
      (Int8) (plpContext.plpLocalDeviceInfo.plpBtBdAddr.nap >> random)));
 return rand;
}

/*****************************************************************************
 * Function:    plpInitTimers
 *
 * Description:  Initialises timer that controls the interval at which outgoing info is transmitted
 *****************************************************************************/
void plpInitTimers (void)
{
 plpContext.plpSendIntervalCounter = 0;
 plpContext.plpStateCounter = 0;
}

/*****************************************************************************
 * Function:    plpStartSendIntervalTimer
 *
 * Description:  Starts timer that controls the interval at which outgoing info is transmitted
 *****************************************************************************/

void plpStartSendIntervalTimer (void)
{
 plpContext.plpSendIntervalCounter ++;

 /* if > 1, counter is already running therefore stop counter and restart it */
 if (1 < plpContext.plpSendIntervalCounter)
 {
  KiStopTimer ( &plpSendIntervalTimer);
 }

 plpSendIntervalTimer.timeoutPeriod = MILLISECONDS_TO_TICKS (PLP_SEND_INTERVAL_TIMER_VALUE);
 plpSendIntervalTimer.myTaskId = PLP_TASK_ID;
 plpSendIntervalTimer.userValue = 0;
 KiStartTimer(&plpSendIntervalTimer);
 plpSendIntervalTimerRunning (TRUE);
 }

/*****************************************************************************
 * Function:    plpStopSendIntervalTimer
 *
```

```
* Description:  Stops timer that controls the interval at which outgoing info is transmitted
 ******************************************************************************/

void plpStopSendIntervalTimer (void)
{
 KiStopTimer(&plpSendIntervalTimer);
 plpSendIntervalTimerRunning (FALSE);
}

/******************************************************************************
* Function:    plpStartStateTimer
*
* Description:  Starts state timeout timer
 ******************************************************************************/

void plpStartStateTimer (void)
{
 plpContext.plpStateCounter ++;

 /* if > 1, counter is already running therefore stop counter and restart it */
 if (1 < plpContext.plpStateCounter)
 {
  KiStopTimer ( &plpStateTimer);
 }

 plpStateTimer.timeoutPeriod = MILLISECONDS_TO_TICKS (PLP_STATE_TIMER_VALUE);
 plpStateTimer.myTaskId = PLP_TASK_ID;
 plpStateTimer.userValue = 0;
 KiStartTimer(&plpStateTimer);
 plpStateTimerRunning (TRUE);
}

/******************************************************************************
* Function:    plpStopStateTimer
*
* Description:  Stops state timeout timer
 ******************************************************************************/

void plpStopStateTimer (void)
{
 KiStopTimer(&plpStateTimer);
 plpStateTimerRunning (FALSE);
}

/******************************************************************************
 * Function:    plpStateTimerRunning
 *
 * Description: TRUE if State timer is running.
 ******************************************************************************/

void plpStateTimerRunning(Boolean plpStateTimerRunning)
{
 plpContext.plpStateTimerRunning = plpStateTimerRunning;
}

/******************************************************************************
 * Function:    plpSendIntervalTimerRunning
 *
```



```
 * Description: TRUE if sendIntervalTimer is running
 ******************************************************************************/

void plpSendIntervalTimerRunning(Boolean plpSendIntervalTimerRunning)
{
 plpContext.plpSendIntervalTimerRunning = plpSendIntervalTimerRunning;
}

/******************************************************************************
* Function:    plpSendInfo
*
* Description:  creates and sends plpOutInfoReq
 ******************************************************************************/
void plpSendInfo (void)
{
 SignalBuffer signalToSend = kiNullBuffer ;

 plpSignalHandled (TRUE);

 KiCreateSignal  (SIG_PLPTX_OUT_INFO_REQ,
          sizeof (PlptxOutInfoReq),
          &signalToSend);

 signalToSend.sig->plptxOutInfoReq.myTaskId = PLP_TASK_ID;
 signalToSend.sig->plptxOutInfoReq.plpFriendlyName = plpContext.plpLocalDeviceInfo.plpFriendlyName;
 signalToSend.sig->plptxOutInfoReq.plpBtBdAddr = plpContext.plpLocalDeviceInfo.plpBtBdAddr;

 memcpy (&signalToSend.sig->plptxOutInfoReq.plpLinkKey,
     &plpContext.plpLocalDeviceInfo.plpLinkKey,
     BT_ENCRYPTION_KEY_SIZE * sizeof (Int8));

 KiSendSignal (PLPTX_TASK_ID,&signalToSend);
}

/******************************************************************************
* Function:    plpInInfo
*
* Description:  Stores info from plpInInfoInd
 ******************************************************************************/

void plpInInfo (SignalBuffer * signalBuffer_p)
{
 SignalBuffer signalToSend = kiNullBuffer ;

      /*store remote device info */

      /*stores BtBdAddr as plpRemoteDeviceInfo.plpBtBdAddr */
 memcpy (&plpContext.plpRemoteDeviceInfo.plpBtBdAddr,
     &signalBuffer_p->sig->plptxInInfoInd.plpBtBdAddr,
     sizeof (BtBdAddr));

 /* stores plpFriendlyName as plpRemoteDeviceInfo.plpFriendlyName */
 memcpy (&plpContext.plpRemoteDeviceInfo.plpFriendlyName,
     &signalBuffer_p->sig->plptxInInfoInd.plpFriendlyName,
     HCIE_07_NAME_SIZE);

 /* stores plpLinkKey as plpRemoteDeviceInfo.plpLinkKey */
 memcpy (&plpContext.plpRemoteDeviceInfo.plpLinkKey,
```

```
    &signalBuffer_p->sig->plptxInInfoInd.plpLinkKey,
    BT_ENCRYPTION_KEY_SIZE * sizeof (Int8));
}

/*****************************************************************************
* Function:   plpLinkInfo
*
* Description: Creates and sends plpLinkInfoInd
*****************************************************************************/
void plpLinkInfo ()
{
 SignalBuffer signalToSend = kiNullBuffer ;

 /* create plp link info signal to send to application */
 KiCreateSignal (SIG_PLP_LINK_INFO_IND,
          sizeof (PlpLinkInfoInd),
          &signalToSend);

 signalToSend.sig->plpLinkInfoInd.myTaskId = PLP_TASK_ID;
 signalToSend.sig->plpLinkInfoInd.plpStatus = PLP_COMMAND_OK;
 signalToSend.sig->plpLinkInfoInd.plpLocalBtBdAddr = plpContext.plpLocalDeviceInfo.plpBtBdAddr;
 signalToSend.sig->plpLinkInfoInd.plpLocalFriendlyName = plpContext.plpLocalDeviceInfo.plpFriendlyName;
 signalToSend.sig->plpLinkInfoInd.plpRemoteBtBdAddr = plpContext.plpRemoteDeviceInfo.plpBtBdAddr;
 signalToSend.sig->plpLinkInfoInd.plpRemoteFriendlyName = plpContext.plpRemoteDeviceInfo.plpFriendlyName;

 if (memcmp(plpContext.plpLocalDeviceInfo.plpLinkKey, plpContext.plpRemoteDeviceInfo.plpLinkKey, 128) >= 0)
 {
  memcpy (&signalToSend.sig->plpLinkInfoInd.plpLinkKey,
       &plpContext.plpLocalDeviceInfo.plpLinkKey,
       BT_ENCRYPTION_KEY_SIZE * sizeof (Int8));
 }
 else
 {
  memcpy (&signalToSend.sig->plpLinkInfoInd.plpLinkKey,
       &plpContext.plpRemoteDeviceInfo.plpLinkKey,
       BT_ENCRYPTION_KEY_SIZE * sizeof (Int8));
 }

 KiSendSignal (TE_TASK_ID,&signalToSend);
}

/*****************************************************************************
* Function:   plptxInInfoRsp
*
* Description: sends plptxInInfoRsp to the plptx task
*****************************************************************************/
void plptxInInfoRsp ()
{
 SignalBuffer signalToSend = kiNullBuffer;

 KiCreateSignal ( SIG_PLPTX_IN_INFO_RSP,
          sizeof (PlptxInInfoRsp),
          &signalToSend);

 signalToSend.sig -> plptxInInfoRsp.myTaskId = PLP_TASK_ID;
 signalToSend.sig -> plptxInInfoRsp.plpStatus = PLP_COMMAND_OK;

 KiSendSignal (PLPTX_TASK_ID, &signalToSend);
```



```
}

/*****************************************************************************
* Function:   plptxOutFinishReq
*
* Description: sends plptxOutFinishReq to the plptx task
*****************************************************************************/
void plptxOutFinishReq ()
{
 SignalBuffer signalToSend = kiNullBuffer;

 KiCreateSignal ( SIG_PLPTX_OUT_FINISH_REQ,
          sizeof (PlptxOutFinishReq),
          &signalToSend);

 signalToSend.sig -> plptxOutFinishReq.myTaskId = PLP_TASK_ID;

 KiSendSignal (PLPTX_TASK_ID, &signalToSend);
```

```
/*****************************************************************************
 *
 * $Workfile:   plp_fnc.h $
 * $Revision:
 *    $Date:
 *
 *****************************************************************************
 *
 * Designed by   : PKR
 * Coded by      :
 * Tested by     : PKR
 *
 *****************************************************************************
 *
 * File Description
 * ----------------
 * Pairing Link Protocol - Main function
 *
 *
 *****************************************************************************/
#if !defined (PLP_FNC_H)

#define PLP_FNC_H

#if !defined (PLPSIGUN_H)
#include "plpsigun.h"
#endif

void plpSwitch          (SignalBuffer *);
void plpStateSwitch     (SignalBuffer *);
void plpInit (void);

#endif
```



```
*****************************************************************************
 *
 * $Workfile:   plpmn_fnc.c $
 * $Revision:
 *    $Date:
 *
 *****************************************************************************
 *
 * Designed by   : PKR
 * Coded by      :
 * Tested by     : PKR
 *
 *****************************************************************************
 *
 * File Description
 * ----------------
 * Pairing Link Protocol - Controlling Task main loop
 *
 *****************************************************************************
 *
 * Revision Details
 * ----------------
 *   $Log:
 *
 *
 *
 *
 *
 *****************************************************************************/

#define MODULE_NAME "PLPMN_FNC"

/*****************************************************************************
 * Include Files                hpsigbas.h      hpsig.h
 *****************************************************************************/

#if defined (HPDEFINE)
# if !defined (HPDEFINE_H)
#   include "hpdefine.h"
# endif
#endif

#if !defined (STRING_H)
# include "string.h"
#endif

#if !defined (KERNEL_H)
# include "kernel.h"
#endif

#if !defined (PLP_SIG_H)
#include "plp_sig.h"
#endif

#if !defined (PLPMN_FNC_H)
#include "plpmn_fnc.h"
#endif
```

```
#if !defined (PLP_FNC_H)
#include "plp_fnc.h"
#endif

#if !defined (PLP_TYP_H)
#include "plp_typ.h"
#endif

#if defined (PLP_TRACE_OUTPUT)
# include "pssignal.h"
# include "emmi_sig.h"
# include "stdio.h"
#endif

/*****************************************************************************
 * Manifest Constants
 *****************************************************************************/

/*****************************************************************************
 * Types
 *****************************************************************************/
 typedef struct PlpmnContextTag
 {
  Boolean          signalHandled;
 } PlpmnContext;

/*****************************************************************************
 * General Variables
 *****************************************************************************/

PlpmnContext       plpmnContext;

#if defined (PLP_TRACE_OUTPUT)
 char traceString[ MAX_TEST_FILE_OUT_STRING ];
#endif

/*****************************************************************************
 * Macros
 *****************************************************************************/


/*****************************************************************************
 * Functional Prototypes
 *****************************************************************************/
static void     PlpTaskExitRoutine  (void);
KI_ENTRY_POINT  PlpTask   (void);
KI_SINGLE_TASK  (PlpTask, PLP_QUEUE_ID, PLP_TASK_ID )



/*****************************************************************************
 *     Global Functions
 *****************************************************************************/

/*****************************************************************************
 * Function:    PlpTaskExitRoutine
 *
 * Description: plp task exit routine
```



```
 *****************************************************************************/

static void PlpTaskExitRoutine( void )
{

}

/*****************************************************************************
 * Function:    PlpTask
 *
 * Description: Main entry point to the PLP task
 *****************************************************************************/

KI_ENTRY_POINT PlpTask()
{
 Boolean keepGoing;
 SignalBuffer signalBuffer = kiNullBuffer;

 /* absorb all signals until a SIG_INITIALISE is received  */
 keepGoing = TRUE;
 while (keepGoing == TRUE)
 {
  KiReceiveSignal (PLP_QUEUE_ID,&signalBuffer);

  if (*(signalBuffer.type) == SIG_INITIALISE)
  {
   keepGoing = FALSE;
  }
  KiDestroySignal (&signalBuffer);
 }

 plpInit ();    /* calls plp initialisation routine*/

 /* Never ending loop */
 keepGoing = TRUE;

 while (keepGoing == TRUE)
 {
  /*mark signal as not handled and destroy at end, then get the next signal*/
  plpSignalHandled(FALSE);

  KiReceiveSignal(PLP_QUEUE_ID,&signalBuffer);

  /*process all signals*/
  plpSwitch (&signalBuffer);

#if defined (DEVELOPMENT_VERSION)
  if (plpmnContext.signalHandled == FALSE)
  {
   char text[100];
   sprintf(&text[0],"Signal %0x Not Handled",*signalBuffer.type);
   DevFail (&text[0]);
   break;
  }
#endif

  KiDestroySignal (&signalBuffer);
 }
```

```
}

/*****************************************************************************
 * Function:    plpSignalHandled
 *                                  plpsig.h    plp_sig.h
 * Description: TRUE if signal is handled.
 *****************************************************************************/

void plpSignalHandled(Boolean signalHandled)
{
 plpmnContext.signalHandled = signalHandled;
}

/*****************************************************************************
 * Function:    plpTraceOutput
 *
 * Description: Sends the buffer to the Genie Trace Output window
 *****************************************************************************/
#if defined PLP_TRACE_OUTPUT

void plpTraceOutput (char *string)
{
 SignalBuffer signalToSend = kiNullBuffer;

 KiCreateSignal (SIG_TEST_FILE_OUT, sizeof(TestFileOut), &signalToSend);
 memcpy (signalToSend.sig->testFileOut.string, string, sizeof(TestFileOut));
 KiSendSignal (TEST_TASK_ID, &signalToSend);
}
#endif

/* How to use PLP_TRACE_OUTPUT... */

/* #if defined (DM_TRACE_OUTPUT)
      sprintf(traceString,"DMMN: DMIN_QU_COM_STATUS Missed.cnf signalID = %x",
           signalBuffer.sig->dminQuComStatus.signalId);
      dmmnTraceOutput(traceString);
#endif */
```



```
*
* $Workfile:   plpmn_fnc.h $
* $Revision:
*    $Date:
*
*****************************************************************************
*
* Designed by    : PKR
* Coded by       :
* Tested by      : PKR
*
*****************************************************************************
*
* File Description
* ----------------
* Pairing Link Protocol - Main function
*****************************************************************************/
#if !defined (PLPMN_FNC_H)
#define PLPMN_FNC_H

#if !defined (PLPSIGUN_H)
#include "plpsigun.h"
#endif

void  plpSignalHandled     (Boolean);

#endif

#if defined (PLP_TRACE_OUTPUT)
void  plpTraceOutput       (char*);
#endif
```

```
/*****************************************************************************
 *
 * $Workfile:  plp_typ.h  $
 * $Revision:
 *     $Date:
 *
 *****************************************************************************
 *
 * File Description :
 *     Globally useful Pairing Link Protocol types/definitions
 *
*****************************************************************************/

#if !defined (PLP_TYP_H)
#define PLP_TYP_H

#if !defined (BT_TYP_H)
#include "bt_typ.h"
#endif

/*****************************************************************************
 * Nested Include Files
 *****************************************************************************/

/*****************************************************************************
 * Manifest Constants
 *****************************************************************************/

#define PLP_STATE_TIMER_VALUE 10000
#define PLP_SEND_INTERVAL_TIMER_VALUE 500   /* was 10 , then 500 then 50*/
#define HCIE_07_NAME_SIZE 248

/*****************************************************************************
 * Global Macros
 *****************************************************************************/


/*****************************************************************************
 * Types used in Prototypes and Globals
 *****************************************************************************/

typedef enum    PlpStatusTag
 {
 PLP_COMMAND_OK,
 PLP_VALID,
 PLP_NOT_VALID,
 PLP_COMMAND_FAIL
 }PlpStatus;

typedef struct  PlpFriendlyNameTag
 {
  Int8          nameLen;
  Char          name[HCIE_07_NAME_SIZE];
 }PlpFriendlyName;

typedef struct  PlpDeviceInfoTag
 {
 BtBdAddr       plpBtBdAddr;
```



```
 PlpFriendlyName   plpFriendlyName;
 Int8              plpLinkKey [BT_ENCRYPTION_KEY_SIZE];
 }PlpDeviceInfo;


/*****************************************************************************
 * Global Static Variables
 *****************************************************************************/

/*****************************************************************************
 * Global Function Prototypes
 *****************************************************************************/

#endif
/* END OF FILE */
```

```
/*************************************************************

 *
 * $Workfile:   plp_sig.h $
 * $Revision:
 *    $Date:
 *
 ****************************************************************************
 *
 * Designed by    : PKR
 * Detailed Design:
 * Coded by       : PKR
 * Tested by      :
 *
 ****************************************************************************
 *
 * File Description
 * ----------------
 *   PLP signal definitions
 *
 ****************************************************************************/


#if !defined (PLP_SIG_H)
#define PLP_SIG_H

#if !defined (SYSTEM_H)
# include "system.h"
#endif

#if !defined (HCI_TYP_H)
#include "hci_typ.h"
#endif

#if !defined (PLP_TYP_H)
#include "plp_typ.h"
#endif

/****************************************************************************
 * Type Definitions
 ****************************************************************************/
typedef struct   PlpRegisterReqTag {
  TaskId          myTaskId;
  Int8            timeout;
}PlpRegisterReq;

typedef struct   PlpStartScanReqTag {
  TaskId          myTaskId;
}PlpStartScanReq;

typedef struct   PlpStartScanCnfTag {
  PlpStatus       plpStatus;
}PlpStartScanCnf;


typedef struct   PlpLinkInfoIndTag{
  TaskId          myTaskId;
  PlpStatus       plpStatus;
```



```
  BtBdAddr        plpLocalBtBdAddr;
  PlpFriendlyName   plpLocalFriendlyName;
  BtBdAddr        plpRemoteBtBdAddr;
  PlpFriendlyName   plpRemoteFriendlyName;
  Int8            plpLinkKey [BT_ENCRYPTION_KEY_SIZE];
}PlpLinkInfoInd;

typedef struct   PlpLinkInfoRspTag{
  TaskId          myTaskId;
  PlpStatus       plpStatus;
}PlpLinkInfoRsp;

#endif
/* END OF FILE */
```

```
/*****************************************************
*******************
 *
 * $Workfile:   plpsigbas.h  $
 * $Revision:
 *     $Date:
 *
 ***************************************************************************
 *
 * File Description
 * ----------------
 *  Signal bases used by PAIRING LINK PROTOCOL Task
 *
 ***************************************************************************/

#if !defined (PLPSIGAS_H)
#define PLPSIGBAS_H

  PLP_SIGNAL_BASE = LAST_CUST_SIGBASE + 0x0100,
  LAST_PLP_SIGBASE  = PLP_SIGNAL_BASE,

#endif
```



```
/*****************************************************

 *
 * $Workfile:   plpsigun.h  $
 * $Revision:
 *     $Date:
 *
 ***************************************************************************
 *
 *  Designed by    : PKR
 *  Coded by       : PKR
 *  Tested by      :
 *
 ***************************************************************************
 *
 * File Description
 * ----------------
 * Header file containing all signal types used in the Pairing Link Protocol, used for
 * debug when inspecting signal unions
 *
 ***************************************************************************

#if !defined (PLPSIGUN_H)

#define PLPSIGUN_H

#if !defined (DMSH_SIG_H)
#include "dmsh_sig.h"
#endif

#if !defined (DMIQ_SIG_H)
#include "dmiq_sig.h"
#endif

#if !defined (DMSC_SIG_H)
#include "dmsc_sig.h"
#endif

#if !defined (DMCN_SIG_H)
#include "dmcn_sig.h"
#endif

#if !defined (DMSO_SIG_H)
#include "dmso_sig.h"
#endif

#if !defined (DML2_SIG_H)
#include "dml2_sig.h"
#endif

#if !defined (DMSP_SIG_H)
#include "dmsp_sig.h"
#endif

#if !defined (DMIN_SIG_H)
#include "dmin_sig.h"
#endif
```

```
#if !defined (DMSD_SIG_H)
#include "dmsd_sig.h"
#endif

#if !defined (L2IF_SIG_H)
#include "l2if_sig.h"
#endif
#endif

#if defined (DM_TRACE_OUTPUT)
#include "emmi_sig.h"

#define SIGNAL TVSIGNAL
#include "sig_def.h"

#include "kernel.h"

#if !defined (PLP_SIG_H)
#include "plp_sig.h"
#endif

#if !defined (PLPTX_SIG_H)
#include "plptx_sig.h"
#endif

union Signal
{

 #if defined (DM_SIGNALS)
 #include "dmsig.h"
 #endif

 #if defined (PLP_SIGNALS)
 #include "plpsig.h"
 #endif

 #if defined (PLPTX_SIGNALS)
 #include "plptxsig.h"
 #endif

 TestFileOut      testFileOut;


 KiInitialiseTask  initialise;
 KiInitialiseTask  kiInitialiseTask;
 KiTimerExpiry     kiTimerExpiry ;

};
#endif
```



```
/*****************************************************************************
 *
 * $Workfile:  plpsig.h $
 * $Revision:
 *    $Date:
 *
 *****************************************************************************
 *
 * File Description
 * ----------------
 *  Pairing Link Protocol signals
 *
 *****************************************************************************/


  SIG_DEF( SIG_PLP_DUMMY = PLP_SIGNAL_BASE,        EmptySignal              plpDummy
)
  SIG_DEF( SIG_PLP_REGISTER_REQ,              PlpRegisterReq           plpRegisterReq          )
  SIG_DEF( SIG_PLP_START_SCAN_REQ,              PlpStartScanReq          plpStartScanReq
)
  SIG_DEF( SIG_PLP_START_SCAN_CNF,              PlpStartScanCnf          plpStartScanCnf
)
  SIG_DEF( SIG_PLP_LINK_INFO_IND,             PlpLinkInfoInd           plpLinkInfoInd          )
```

```
/*****************************************************************************
 *
 * $Workfile:  plptxmn_fnc.c  $
 * $Revision:
 *    $Date:
 *
 *****************************************************************************
 *
 * Designed by   :PKR
 * Coded by      :
 * Tested by     :PKR
 *
 *****************************************************************************
 *
 * File Description
 * ----------------
 * Pairing Link Protocol - Controlling Task main loop
 *
 *****************************************************************************/
#define MODULE_NAME "PLPTXMN_FNC"

/*****************************************************************************
 * Include Files
 *****************************************************************************/

#if defined (HPDEFINE)
#if !defined (HPDEFINE_H)
#include "hpdefine.h"
#endif
#endif

#if !defined (STRING_H)
#include "string.h"
#endif

#if !defined (KERNEL_H)
#include "kernel.h"
#endif

#if !defined (PLPTXMN_FNC_H)
#include "plptxmn_fnc.h"
#endif

#if defined (PLPTX_TRACE_OUTPUT)
# include "stdio.h"
#endif

# include "stdio.h"

#if !defined (PLPTXBU_FNC_C)
#include "plptxbu_fnc.c"
#endif

#if !defined (PLP_TYP_H)
#include "plp_typ.h"
#endif

#if !defined (PLPTXBU_TYP_H)
```



```
#include "plptxbu_typ.h"
#endif

#if defined (PLPTX_TRACE_OUTPUT)
# include "pssignal.h"
# include "emmi_sig.h"
# include "stdio.h"
#endif


/*****************************************************************************
 * Manifest Constants
 *****************************************************************************/

#define PLP_SIGNALS_COUNTS 10

/*****************************************************************************
 * Types
 *****************************************************************************/

/*****************************************************************************
 * Variables
 *****************************************************************************/
#if defined (PLPTX_TRACE_OUTPUT)
char traceString[ MAX_TEST_FILE_OUT_STRING ];
#endif

/*****************************************************************************
 * Timer variables
 *****************************************************************************/

KiTimer         plptxStartSequenceTimer;

/*****************************************************************************
 * Macros
 *****************************************************************************/

/*****************************************************************************
 * Functional Prototypes
 *****************************************************************************/

void plptxOutInfoReq (PlptxOutInfoReq *);

static void     PLPTXTaskExitRoutine          (void);
KI_ENTRY_POINT  PLPTXTask                     (void);
KI_SINGLE_TASK  (PLPTXTask, PLPTX_QUEUE_ID, PLPTX_TASK_ID )
void plptxStateSwitch  (SignalBuffer*);
void sendInfoTypeByte (void);
void plptxBusAckReq (void);
void plptxSendOutFinishReq(void);
void plptxDisconnectedState(SignalBuffer *signalBuffer_p);
void plptxConnectedState(SignalBuffer *signalBuffer_p);
void plptxStartSequenceReq (PlptxStartSequenceReq *);
void plptxStartSequence2Req (PlptxStartSequence2Req *);

/* for the timer */
void plptxInitStartSequenceTimer (void);
```

```
/*****************************************************************************
 *    Global Functions
 *****************************************************************************/

/*****************************************************************************
 * Function:   PLPTaskExitRoutine
 *
 * Description: app task exit routine
 *****************************************************************************/

static void PLPTXTaskExitRoutine( void )
{

}

/*****************************************************************************
 * Function:   PlptxTask
 *
 * Description: Main entry point to the device manager task
 *****************************************************************************/

KI_ENTRY_POINT PLPTXTask()
{
 Boolean keepGoing;
 SignalBuffer signalBuffer = kiNullBuffer;
 SignalBuffer signalToSend = kiNullBuffer;
 DmshRegisterApplicationReq * dmshRegisterApplicationReq_p;

#if defined (DEVELOPMENT_VERSION)
 sprintf(traceString,"PLPTX: entry point reached");
 plptxTraceOutput(traceString);
#endif

 /* absorb all signals until a SIG_INITIALISE is received  */
 keepGoing = TRUE;
 while (keepGoing == TRUE)
 {
  KiReceiveSignal (PLPTX_QUEUE_ID,&signalBuffer);
  if (*(signalBuffer.type) == SIG_INITIALISE)
  {
   keepGoing = FALSE;
  }
  KiDestroySignal (&signalBuffer);
 }

 plptxContext.plptxState = DISCONNECTED;  /* set initial tx state */
 plptxContext.start1Received = FALSE;

 KiCreateSignal          (SIG_DMSH_REGISTER_APPLICATION_REQ,
           sizeof (DmshRegisterApplicationReq),
           &signalToSend);

 dmshRegisterApplicationReq_p = &signalToSend.sig->dmshRegisterApplicationReq;

 memset (dmshRegisterApplicationReq_p,0,sizeof(DmshRegisterApplicationReq));

 dmshRegisterApplicationReq_p->taskId = PLPTX_TASK_ID;
```



```
 dmshRegisterApplicationReq_p->registerAsApplication= TRUE;
 KiSendSignal (DM_TASK_ID,&signalToSend);

 plptxBusInit ();      /* calls plptx initialisation routine*/

 /* set signalcount to zero */
 plptxContext.plpSignalCount = 0;

 /* Never ending loop */
 keepGoing = TRUE;
 while (keepGoing == TRUE)
 {
  /*mark signal as not handled and destroy at end, then get the next signal*/
  plptxSignalHandled(FALSE);

  /*there are no signal no the internal unit queue, so
   remove the next signal from the external queue*/
  KiReceiveSignal(PLPTX_QUEUE_ID,&signalBuffer);
  DevAssert ((signalBuffer.type)!=PNULL);

  /*process all signals*/
  plptxStateSwitch (&signalBuffer);

#if defined (DEVELOPMENT_VERSION)
  if (plptxContext.signalHandled == FALSE)
  {
   char text[100];
   sprintf(&text[0],"Signal %0x Not Handled",*signalBuffer.type);
   DevFail (&text[0]);
   break;
  }
#endif

  KiDestroySignal (&signalBuffer);
 }
}

/*****************************************************************************
 * Function:   plptxSignalHandled
 *
 * Description: TRUE if signal is handled.
 *****************************************************************************/

void plptxSignalHandled(Boolean signalHandled)
{
 plptxContext.signalHandled = signalHandled;
}


/*****************************************************************************
 * Function:   plptxStateSwitch
 *
 * Description:  Main switch for PLPTX task
 *****************************************************************************/
void plptxStateSwitch (SignalBuffer *signalBuffer_p)
{
 switch (plptxContext.plptxState)
 {
```

```
  case DISCONNECTED:
   plptxDisconnectedState(signalBuffer_p);
   break;

  case CONNECTED:
   plptxConnectedState(signalBuffer_p);
   break;

  default:
   DevFail ("incorrect txState");
   break;
 }
}

/*****************************************************************************
 * Function:    plptxDisconnectedState
 *
 * Description:  Tx disconnected state
 *****************************************************************************/
void plptxDisconnectedState(SignalBuffer *signalBuffer_p)
{
 SignalBuffer  signalToSend = kiNullBuffer;

 plptxSignalHandled(TRUE);

 switch (*(signalBuffer_p->type))
 {
  case SIG_DMSH_REGISTER_APPLICATION_CNF:

   DevAssert (signalBuffer_p->sig->dmshRegisterApplicationCnf.comStatus == COMMAND_OK);

   if (signalBuffer_p->sig->dmshRegisterApplicationCnf.comStatus != COMMAND_OK)
   {
    plptxSignalHandled (FALSE);
   }

   break;

  case SIG_TIMER_EXPIRY:

   /* If it's the StartSequenceTimer that has expired */
   if (signalBuffer_p->sig->kiTimerExpiry.timerId == plptxStartSequenceTimer.timerId)
   {
    if (plptxContext.plptxStartSequenceTimerRunning == TRUE)
    {
     #if defined (PLP_TRACE_OUTPUT)
      sprintf(traceString,"PLPTX: StartSequence TIMER_EXPIRY occurred in PLPTX DISCONNECTED state");
      plpTraceOutput(traceString);
     #endif

     /* activate start sequence (part 1 of 2)*/
     KiCreateSignal     (SIG_PLPTX_START_SEQUENCE_REQ,
                 sizeof (PlptxStartSequenceReq),
                 &signalToSend);

     signalToSend.sig->plptxStartSequenceReq.myTaskId = PLPTX_TASK_ID;
     signalToSend.sig->plptxStartSequenceReq.localBtBdAddr = plptxContext.localDeviceId;
```



```
     KiSendSignal  (PLPTX_TASK_ID, &signalToSend);
    }
    else
    {
     /* do nothing as the timer has already been stopped - plptxContext.plpStartSequenceTimerRunning = FALSE */

#if defined (PLP_TRACE_OUTPUT)
     sprintf(traceString,"PLPTX: StartSequence TIMER_EXPIRY ignored as timer stopped");
     plpTraceOutput(traceString);
#endif
    }
   }
   else
   {
    /* signal for use elsewhere let it through */
    plptxSignalHandled(FALSE);
   }
   break;


  case SIG_PLPTX_START_SEQUENCE_REQ:
   /* Start StartSequenceTimer */
   plptxStartStartSequenceTimer ();

   memcpy (&plptxContext.localDeviceId, &signalBuffer_p->sig->plptxStartSequenceReq.localBtBdAddr, sizeof
(BtBdAddr));
   plptxStartSequenceReq(&signalBuffer_p->sig->plptxStartSequenceReq);
   break;


  case SIG_PLPTX_START_SEQUENCE2_REQ:
#if defined (DEVELOPMENT_VERSION)
   sprintf(traceString,"PLPTX: received start sequence2 req");
   plptxTraceOutput(traceString);
#endif
   plptxStartSequence2Req(&signalBuffer_p->sig->plptxStartSequence2Req);

   plptxContext.start1Received = TRUE;
   break;


  case SIG_PLPTX_START_SEQUENCE_CNF:
   if (plptxContext.start1Received == TRUE)
   {
    /* send plpStartSequenceCnf to PLPTX task*/
    KiCreateSignal      (SIG_PLPTX_START_SEQUENCE_CNF,
                sizeof (PlptxStartSequenceCnf),
                &signalToSend);

    signalToSend.sig->plptxStartSequenceCnf.myTaskId = PLPTX_TASK_ID;

    KiSendSignal  (PLP_TASK_ID, &signalToSend);

    plptxStopStartSequenceTimer ();

    /* handshaking complete go to CONNECTED state */
    plptxContext.plptxState = CONNECTED;
   }
```

```
    else
    {
     sprintf(traceString,"PLPTX: ignored as start1 (AAA) not received, before rx Start2 (ABB)");
     plptxTraceOutput(traceString);
    }
   break;


  case SIG_PLPTX_OUT_INFO_REQ:
   sprintf(traceString,"PLPTX: hello)");
   plptxTraceOutput(traceString);
   break;


  default:
   plptxSignalHandled (FALSE);
   break;
 }
}
/*****************************************************************************
 * Function:    plptxConnectedState
 *
 * Description:  Tx connected state
 *****************************************************************************/
void plptxConnectedState(SignalBuffer *signalBuffer_p)
{
 plptxSignalHandled(TRUE);

 switch (*(signalBuffer_p->type))
 {
  /* left over signals from previous state*/
  case SIG_PLPTX_START_SEQUENCE2_REQ:
   plptxStartSequence2Req(&signalBuffer_p->sig->plptxStartSequence2Req);
#if defined (DEVELOPMENT_VERSION)
   sprintf(traceString,"PLPTX: now connected - sent START_SEQUENCE2_REQ anyway");
   plptxTraceOutput(traceString);
#endif
   break;

  case SIG_PLPTX_START_SEQUENCE_CNF:
#if defined (DEVELOPMENT_VERSION)
   sprintf(traceString,"PLPTX: now connected - ignored START_SEQUENCE_CNF");
   plptxTraceOutput(traceString);
#endif
   break;

  case SIG_PLPTX_START_SEQUENCE_REQ:
#if defined (DEVELOPMENT_VERSION)
   sprintf(traceString,"PLPTX: now connected - ignored START_SEQUENCE_REQ");
   plptxTraceOutput(traceString);
#endif
   break;


  /* signals for this state */
  case SIG_PLPTX_OUT_INFO_REQ:
   plptxOutInfoReq(&signalBuffer_p->sig->plptxOutInfoReq);
   break;
```



```
  case SIG_DMSH_REGISTER_APPLICATION_CNF:
   DevAssert (signalBuffer_p->sig->dmshRegisterApplicationCnf.comStatus == COMMAND_OK);
   break;


  case SIG_PLPTX_IN_INFO_RSP:
   /* task needs to send an ACK signal to t'other pc */
   plptxBusAckReq ();
   break;


  case SIG_PLPTX_OUT_FINISH_REQ:
   plptxSendOutFinishReq();
   break;


  default:
   plptxSignalHandled(FALSE);
   break;
 }
}
/*****************************************************************************
 * Function: plptxStartSequenceReq
 *
 * Description: process the plptxStartSequenceReq signal
 *****************************************************************************/
void plptxStartSequenceReq (PlptxStartSequenceReq * plptxStartSequenceReq_p)
{
 SignalBuffer busSignal = kiNullBuffer ;
 Int8*       packetBuffer;

 KiCreateSignal          (SIG_PLPTX_BUS_WRITE_DATA_REQ,
           sizeof (PlptxBusWriteDataReq),
           &busSignal);

 packetBuffer = (Int8*) & (busSignal.sig -> plptxBusWriteDataReq.txData);

 /* fill signal */
 /* header */
 PUT_INT8 (0, packetBuffer, PRE_AMBLE_BYTE);
 PUT_INT8 (1, packetBuffer, PRE_AMBLE_BYTE);
 PUT_INT8 (2, packetBuffer, PRE_AMBLE_BYTE);
 PUT_BDADDR(3, packetBuffer, plptxContext.localDeviceId);

 PUT_INT8((PLPTX_BUS_HEADER_SIZE), packetBuffer, START_TYPE);   /* signal type */
 PUT_INT8 ((SIZE_TO_SIGNAL_NAME), packetBuffer, START_SIGNAL_NAME);
 PUT_INT8 (SIZE_TO_SIGNAL, packetBuffer, START_SIGNAL);

 busSignal.sig -> plptxBusWriteDataReq.txDataSize = (PLPTX_BUS_START_SEQUENCE_SIZE +
PLPTX_BUS_HEADER_SIZE);

 plptxBusWriteData (&busSignal);
 KiDestroySignal ( &busSignal ) ;

}
```

```
/*****************************************************************************
 * Function: plptxStartSequence2Req
 *
 * Description: process the plptxStartSequence2Req signal
 *****************************************************************************/
void plptxStartSequence2Req (PlptxStartSequence2Req * plptxStartSequence2Req_p)
{
 SignalBuffer busSignal = kiNullBuffer ;
 Int8*       packetBuffer;

 KiCreateSignal          (SIG_PLPTX_BUS_WRITE_DATA_REQ,
             sizeof (PlptxBusWriteDataReq),
             &busSignal);

 packetBuffer = (Int8*) & (busSignal.sig -> plptxBusWriteDataReq.txData);

 /* fill signal */
 /*header */
 PUT_INT8 (0, packetBuffer, PRE_AMBLE_BYTE);
 PUT_INT8 (1, packetBuffer, PRE_AMBLE_BYTE);
 PUT_INT8 (2, packetBuffer, PRE_AMBLE_BYTE);
 PUT_BDADDR(3, packetBuffer, plptxContext.localDeviceId);

 PUT_INT8((PLPTX_BUS_HEADER_SIZE), packetBuffer, START_TYPE);   /* signal type */
 PUT_INT8 ((SIZE_TO_SIGNAL_NAME), packetBuffer, START2_SIGNAL_NAME);
 PUT_INT8 (SIZE_TO_SIGNAL, packetBuffer, START2_SIGNAL);

 busSignal.sig -> plptxBusWriteDataReq.txDataSize = (PLPTX_BUS_START_SEQUENCE_SIZE +
PLPTX_BUS_HEADER_SIZE);

 plptxBusWriteData (&busSignal);
 KiDestroySignal ( &busSignal ) ;

}
/*****************************************************************************
 * Function: plptxOutInfoReq
 *
 * Description: process the plpOutInfoReq signal
 *****************************************************************************/

void plptxOutInfoReq (PlptxOutInfoReq * plptxOutInfoReq_p)
{
 int friendlyNameCount;
 int linkKeyCount;

 SignalBuffer busSignal = kiNullBuffer ;
 Int8*       packetBuffer;

 KiCreateSignal          (SIG_PLPTX_BUS_WRITE_DATA_REQ,
             sizeof (PlptxBusWriteDataReq),
             &busSignal);

 packetBuffer = (Int8*) & (busSignal.sig -> plptxBusWriteDataReq.txData);


 /* fill signal */
 /*header */
 PUT_INT8 (0, packetBuffer, PRE_AMBLE_BYTE);
```



```
 PUT_INT8 (1, packetBuffer, PRE_AMBLE_BYTE);
 PUT_INT8 (2, packetBuffer, PRE_AMBLE_BYTE);
 PUT_BDADDR(3, packetBuffer, plptxContext.localDeviceId);

 PUT_INT8(PLPTX_BUS_HEADER_SIZE, packetBuffer, INFO_TYPE);   /* signal type */
 PUT_INT8 ((SIZE_TO_SIGNAL_NAME), packetBuffer, PLPTX_BUS_OUT_INFO);

 PUT_BDADDR ((SIZE_TO_BDADDR), packetBuffer, plptxOutInfoReq_p -> plpBtBdAddr);

 PUT_INT8((SIZE_TO_NAME_LEN), packetBuffer,  plptxOutInfoReq_p -> plpFriendlyName.nameLen);

 for (friendlyNameCount = 0; friendlyNameCount <248; friendlyNameCount ++)
 {
  PUT_INT8 ((SIZE_TO_NAME + friendlyNameCount),packetBuffer, plptxOutInfoReq_p -> plpFriendlyName.name
[friendlyNameCount]);
 }


 for (linkKeyCount = 0; linkKeyCount < 16; linkKeyCount++)
 {
  PUT_INT8 ((SIZE_TO_LINK_KEY+ linkKeyCount), packetBuffer, plptxOutInfoReq_p -> plpLinkKey
[linkKeyCount]);
 }

 busSignal.sig -> plptxBusWriteDataReq.txDataSize = (PLPTX_BUS_OUT_INFO_DATA_SIZE +
PLPTX_BUS_HEADER_SIZE);

 plptxBusWriteData (&busSignal);
 KiDestroySignal ( &busSignal ) ;
}

/*****************************************************************************
 * Function: plptxBusAckReq
 *
 * Description: process the plptxBusAckReq signal
 *****************************************************************************/

void plptxBusAckReq ()
{
 SignalBuffer busSignal = kiNullBuffer ;
 Int8*       packetBuffer;

 KiCreateSignal          (SIG_PLPTX_BUS_WRITE_DATA_REQ,
             sizeof (PlptxBusWriteDataReq),
             &busSignal);

 packetBuffer = (Int8*) & (busSignal.sig -> plptxBusWriteDataReq.txData);

 /*header */
 PUT_INT8 (0, packetBuffer, PRE_AMBLE_BYTE);
 PUT_INT8 (1, packetBuffer, PRE_AMBLE_BYTE);
 PUT_INT8 (2, packetBuffer, PRE_AMBLE_BYTE);
 PUT_BDADDR(3, packetBuffer, plptxContext.localDeviceId);

 PUT_INT8 (SIZE_TO_TYPE, packetBuffer, ACK_TYPE);
 PUT_INT8 (SIZE_TO_SIG_NAME, packetBuffer, PLPTX_BUS_ACK);
 PUT_INT8 (SIZE_TO_SIGNAL_BEING_ACKED, packetBuffer, PLPTX_BUS_OUT_INFO);
```

```
 busSignal.sig -> plptxBusWriteDataReq.txDataSize = (PLPTX_BUS_ACK_TYPE_SIZE +
PLPTX_BUS_HEADER_SIZE);
 plptxBusWriteData (&busSignal);
 KiDestroySignal (&busSignal);
}

/*****************************************************************************
 * Function: plptxSendOutFinishReq
 *
 * Description: send the OutFinishReq signal to the other device
 *****************************************************************************/
void plptxSendOutFinishReq ()
{
 SignalBuffer busSignal = kiNullBuffer ;
 Int8*       packetBuffer;

 KiCreateSignal          (SIG_PLPTX_BUS_WRITE_DATA_REQ,
            sizeof (PlptxBusWriteDataReq),
            &busSignal);

 packetBuffer = (Int8*) & (busSignal.sig -> plptxBusWriteDataReq.txData);

 /* fill signal */
 /* header */
 PUT_INT8 (0, packetBuffer, PRE_AMBLE_BYTE);
 PUT_INT8 (1, packetBuffer, PRE_AMBLE_BYTE);
 PUT_INT8 (2, packetBuffer, PRE_AMBLE_BYTE);
 PUT_BDADDR(3, packetBuffer, plptxContext.localDeviceId);

 /* main signal */
 PUT_INT8 (SIZE_TO_TYPE, packetBuffer, ACK_TYPE);
 PUT_INT8 (SIZE_TO_SIG_NAME, packetBuffer, PLPTX_BUS_ACK);
 PUT_INT8 (SIZE_TO_SIGNAL_BEING_ACKED, packetBuffer, PLPTX_FINISH_REQ_ACK);

 busSignal.sig -> plptxBusWriteDataReq.txDataSize = (PLPTX_BUS_ACK_TYPE_SIZE +
PLPTX_BUS_HEADER_SIZE);
 plptxBusWriteData (&busSignal);
 KiDestroySignal (&busSignal);
}

/*****************************************************************************
 * Function:    sendInfoTypeByte
 *
 * Description: sends Info type Byte
 *****************************************************************************/
void sendInfoTypeByte (void)
{
 SignalBuffer busSignal = kiNullBuffer ;
 Int8*       packetBuffer;

 KiCreateSignal          (SIG_PLPTX_BUS_WRITE_DATA_REQ,
            sizeof (PlptxBusWriteDataReq),
            &busSignal);

 packetBuffer = (Int8*) & (busSignal.sig -> plptxBusWriteDataReq.txData);


 /* fill signal */
```



```
 PUT_INT8(0, packetBuffer, INFO_TYPE);   /* signal type */
 busSignal.sig -> plptxBusWriteDataReq.txDataSize = 1;

 plptxBusWriteData (&busSignal);

 KiDestroySignal ( &busSignal ) ;
}

/*****************************************************************************
 * Function:    plptxInit
 *
 * Description: Initialise plptx task
 *****************************************************************************/
void plptxInit(void)
{
 SignalBuffer signalToSend = kiNullBuffer ;
 DmshRegisterApplicationReq * dmshRegisterApplicationReq_p;

 KiCreateSignal          (SIG_DMSH_REGISTER_APPLICATION_REQ,
            sizeof (DmshRegisterApplicationReq),
            &signalToSend);

 dmshRegisterApplicationReq_p = &signalToSend.sig->dmshRegisterApplicationReq;

 memset (dmshRegisterApplicationReq_p,0,sizeof(DmshRegisterApplicationReq));

 dmshRegisterApplicationReq_p->taskId = PLPTX_TASK_ID;
 dmshRegisterApplicationReq_p->registerAsApplication= TRUE;

 KiSendSignal (DM_TASK_ID,&signalToSend);
}

/*****************************************************************************
* Function:    plptxInitStartSequenceTimer
*
* Description:  Initialises timer that controls the interval at which start sequence is transmitted
 *****************************************************************************/
void plptxInitStartSequenceTimer (void)
{
 plptxContext.plptxStartSequenceCounter = 0;
}

/*****************************************************************************
* Function:    plptxStartStartSequenceTimer
*
* Description:  Starts StartSequenceTimer
 *****************************************************************************/

void plptxStartStartSequenceTimer (void)
{
 plptxContext.plptxStartSequenceCounter ++;

 /* if > 1, counter is already running therefore stop counter and restart it */
 if (1 < plptxContext.plptxStartSequenceCounter)
 {
  KiStopTimer ( &plptxStartSequenceTimer);
 }
```

```
 plptxStartSequenceTimer.timeoutPeriod = MILLISECONDS_TO_TICKS
(PLPTX_START_SEQUENCE_TIMER_VALUE);
 plptxStartSequenceTimer.myTaskId = PLPTX_TASK_ID;
 plptxStartSequenceTimer.userValue = 0;
 KiStartTimer(&plptxStartSequenceTimer);
 plptxStartSequenceTimerRunning (TRUE);
}

/*****************************************************************************
* Function:    plptxStopStartSequenceTimer
*
* Description:  Stops StartSequence timer
*****************************************************************************/

void plptxStopStartSequenceTimer (void)
{
 KiStopTimer(&plptxStartSequenceTimer);
 plptxStartSequenceTimerRunning (FALSE);
}

/*****************************************************************************
 * Function:    plptxStartSequenceTimerRunning
 *
 * Description: TRUE if StartSequenceTimer is running.
 *****************************************************************************/

void plptxStartSequenceTimerRunning(Boolean plptxStartSequenceTimerRunning)
{
 plptxContext.plptxStartSequenceTimerRunning = plptxStartSequenceTimerRunning;
}



#if defined PLPTX_TRACE_OUTPUT
/*****************************************************************************
 * Function:    plptxTraceOutput
 *
 * Description: Sends the buffer to the Genie Trace Output window
 *****************************************************************************/

void plptxTraceOutput (char *string)
{
 SignalBuffer signalToSend = kiNullBuffer;

 KiCreateSignal (SIG_TEST_FILE_OUT, sizeof(TestFileOut), &signalToSend);
 memcpy (signalToSend.sig->testFileOut.string, string, sizeof(TestFileOut));
 KiSendSignal (TEST_TASK_ID, &signalToSend);
}
#endif
```



```
/*****************************************************************************
 *
 * $Workfile:  plptxmn_fnc.h $
 * $Revision:
 *    $Date:
 *
 *****************************************************************************
 *
 *  Designed by   : PKR
 *  Coded by      :
 *  Tested by     : PKR
 *
 *****************************************************************************
 *
 * File Description
 * ----------------
 *  Transport Layer - main functions
 *
 *****************************************************************************/

#if !defined (PLPTXMN_FNC_H)
# define PLPTXMN_FNC_H

#if 0
#if !defined (PLPTXSIGUN_H)
#include "plptxsigun.h"
#endif
#endif

#if !defined (BT_TYP_H)
#include "bt_typ.h"
#endif

typedef enum PlptxStateTag
{
 DISCONNECTED,
 CONNECTED
}PlptxState;

typedef struct   PlptxContextTag
{
 Boolean          signalHandled;
 Int8           plpSignalCount;
 Int32           plptxStartSequenceCounter;
 Boolean          plptxStartSequenceTimerRunning;

 PlptxState       plptxState;
 Boolean          busConnected;
 BtBdAddr         rxDeviceId;  /* ID just received */
 BtBdAddr         remoteDeviceId;   /* used for verifying that signal received is not signal transmitted by the same
device */
 BtBdAddr         localDeviceId;
 Boolean          start1Received;
 Boolean          start1Sent;

}PlptxContext;

PlptxContext       plptxContext;
```

```
 void plptxInit (void);
 void  plptxSignalHandled        (Boolean);
 void  plptxDoNotDestroy         (Boolean);

void plptxStartStartSequenceTimer (void);
void plptxStopStartSequenceTimer (void);
void plptxStartSequenceTimerRunning (Boolean);

 #if defined (PLPTX_TRACE_OUTPUT)
   void  plptxTraceOutput         (char*);
 #endif

#endif
```



```
/*****************************************************************************
 *
 * $Workfile:  plptxbu_fnc.c  $
 * $Revision:
 *    $Date:
 *
 *****************************************************************************
 *
 * Designed by   : PKR
 * Coded by      :
 * Tested by     : PKR
 *
 *****************************************************************************
 *
 * File Description
 * ----------------
 * Transport Layer, bus task - main functions
 *
 *****************************************************************************/
/*#define  DEBUG_PLPTX   */

#include <stdio.h>
#include <windows.h>
#include <winbase.h>
#include <windowsx.h>
#include <malloc.h>

#include "stdlib.h"

#if !defined (MA_TYP_H)
#  include "ma_typ.h"
#endif

#if !defined (TM_TYP_H)
#  include "tm_typ.h"
#endif

#if defined(HPS_ON_WINDOWS)
#  if !defined (GWCOMERR_H)
#    include "gwcomerr.h"
#  endif
#endif

#if !defined (STRING_H)
#include "string.h"
#endif


#if !defined (PLPTX_SIG_H)
#include "plptx_sig.h"
#endif


#if !defined (PLPTX_TYP_H)
#include "plptx_typ.h"
#endif
```

```
#if !defined (PLPTXSIGUN_H)
#include "plptxsigun.h"
#endif


#if !defined (PLPTXMN_FNC_H)
#include "plptxmn_fnc.h"
#endif

#if !defined (PLPTXBU_FNC_H)
#include "plptxbu_fnc.h"
#endif

#if !defined (KERNEL_H)
#include "kernel.h"
#endif

#if defined (HPDEFINE)
#if !defined (HPDEFINE_H)
#include "hpdefine.h"
#endif
#endif

#if !defined (PLPTXBU_TYP_H)
#include "plptxbu_typ.h"
#endif

#if defined (PLPTX_TRACE_OUTPUT)
char traceString[ MAX_TEST_FILE_OUT_STRING ];
#endif



/*****************************************************************************
* Variables
*
*
*****************************************************************************/



/*****************************************************************************
* Types
*
*
*****************************************************************************/
typedef enum PlpbuRxStateTag
{
 PLPTX_BUS_START,
 PLPTX_BUS_RX_REMOTE_DEVICE_ID,
 PLPTX_BUS_RX_PACKET_TYPE,
 PLPTX_BUS_RX_SIGNAL
}PlpbuRxState;


typedef struct PlpbuContextTag
{
 PlpbuRxState     rxState;     /*State of the receiver */
```



```
 Boolean        rxActive; /* TRUE if active, FALSE if no longer connected */
 Char          rxBuffer [ 280 ];  /*Initial buffer for packet type and enough for length data */
 Char*          rxPtr;  /* Pointer to where received data should be written */
 Int16         rxLenReceived; /*Length of current received data */
 Int16         rxLenWaiting;  /*Length of data we are waiting to receive */
 CRITICAL_SECTION  qLock;     /* used for locking the send Queue */
 HANDLE          txSignal;  /* handle to event used to signal more data to send */
 HANDLE          wrEvent;   /* write event for overlapped write */
 HANDLE          rdEvent;   /*Read event for overlapped read */
 HANDLE          txHandle;  /* handle to thread running the transmitter */
 HANDLE          rxHandle;  /* handle to thread running the receiver */
 HANDLE          ioHandle;
 KiUnitQueue     cmdQueue;  /* Queue of commmand packets to send */
 Int16         txLenToSend; /* total number of bytes to transmit */
 Int16         txLenSent;   /* number of bytes already sent from this packet */
 Int8          startByteCounter; /* number of start bytes received */
} PlpbuContext;

PRIVATE PlpbuContext  plpbuContext;

/*****************************************************************************
* Function Protoypes
*
*
*****************************************************************************/
DWORD WINAPI  transmitPacket  ( LPVOID );
DWORD WINAPI  plpbuReceivePacket  (LPVOID);
void plptxBusWriteData (SignalBuffer *signalBuffer);
static void plpbuLockQueues(void);
static void plpbuUnlockQueues(void);
void plpbuCreateThreads (void);
static void plpbuSignalMoreTxData (void);
static void plpbuPlaceTxDataOnQueue (SignalBuffer *signalBuffer);
static Boolean plpbuTakeTxDataFromQueue (SignalBuffer  *signalBuffer);
static HANDLE plptxbuOpenPcPort ( void );
void WriterGeneric(Int8 * lpBuf, DWORD dwToWrite);
WtErr WINAPI ReadGeneric( Int8* lpBuf, Int16 dwToRead, Int32 *readBytes );
void plptxBusInit (void);
static void plpbuProcessRxData ( Int16 rxLen);



void plptxBusInit (void)
{
 int rxBufCount; /* for clearing the rxBuffer */
/*****************************************************************************/
 /* initialise the Tx lock */
 InitializeCriticalSection (&plpbuContext.qLock);

 /* Win32 function */
 plpbuContext.txSignal = CreateEvent ( NULL,/* Security attributes */
                        TRUE, /* Manual Reset */
                        FALSE, /* Initial State */
                        NULL  /* name */
                        );
```

```
plpbuContext.wrEvent = CreateEvent ( NULL,/* Security attributes */
                        TRUE, /* Manual Reset */
                        FALSE, /* Initial State */
                        NULL   /* name */
                        );


plpbuContext.rdEvent = CreateEvent ( NULL,/* Security attributes */
                        TRUE, /* Manual Reset */
                        FALSE, /* Initial State */
                        NULL   /* name */
                        );

if (plpbuContext.rdEvent == NULL)
{
 DevFail("Error creating rdEvent");
}
else
{
 sprintf(traceString,"PLPTX: rdEvent created successfully");
 plptxTraceOutput(traceString);
}

if (plpbuContext.wrEvent == NULL)
{
 DevFail("Error creating wrEvent");
}

if (plpbuContext.txSignal == NULL)
{
 DevFail("Error creating txSignal Event");
}
/*****************************************************************************/

/* set all handles to invalid */
plpbuContext.ioHandle = INVALID_HANDLE_VALUE;
plpbuContext.txHandle = INVALID_HANDLE_VALUE;
plpbuContext.rxHandle = INVALID_HANDLE_VALUE;

/*clear queue */
plpbuLockQueues ();
KiFlushQueue (&plpbuContext.cmdQueue );
plpbuUnlockQueues ();

/* set up Rx State */
plpbuContext.rxActive = TRUE;
plpbuContext.rxLenWaiting = 1;
plpbuContext.rxLenReceived = 0;
plpbuContext.rxState = PLPTX_BUS_START;
plpbuContext.startByteCounter = 0;

/* clear the rxBuffer */
for (rxBufCount = 0; rxBufCount<280; rxBufCount++)
{
 plpbuContext.rxBuffer [rxBufCount] = 0 ;
}

plpbuContext.rxPtr = plpbuContext.rxBuffer;
```



```
 /* set up the tx state */
 plpbuContext.txLenToSend = 0;
 plpbuContext.txLenSent = 0;
 ResetEvent (plpbuContext.txSignal);

 /* open the pc port */
 plpbuContext.ioHandle = plptxbuOpenPcPort ();

 plpbuCreateThreads ();

}

/*****************************************************************************
* plpbuOpenPcPort                                                            *
*                                                                            *
*  Opens the port                                                            *
*                                                                            *
*****************************************************************************/
static HANDLE plptxbuOpenPcPort ( void )
{
 HANDLE handle = INVALID_HANDLE_VALUE;
 DCB dcb;
 BOOL          portReady;

 COMMTIMEOUTS    timeoutsDefault;

 /* To open the port */
 if ( plpbuContext.ioHandle == INVALID_HANDLE_VALUE )
 {
  handle = CreateFile (PC_COM_PORT,
           GENERIC_READ | GENERIC_WRITE,
           0,                /* share Port */
           NULL,             /* No Security */
           OPEN_EXISTING,    /* How to Create */
           FILE_FLAG_OVERLAPPED,          /* File Attributes - No overlapping */
           NULL        /* Handle of file with attributes to copy */
           );
 }


 /* Get current Device Control Block Settings */
 GetCommState (handle, &dcb);

 /* fill in the dcb */

 dcb.DCBlength = sizeof (dcb);      /* sizeof(DCB)*/
 dcb.BaudRate = PC_BUS_BAUD_RATE;    /* current baud setting*/
 dcb.fBinary = TRUE;               /* binary mode, no EOF check*/
 dcb.fParity = FALSE;              /* enable parity checking*/
 dcb.fOutxCtsFlow = FALSE;/* was TRUE */          /* CTS output flow control*/
 dcb.fOutxDsrFlow = FALSE;         /* DSR output flow control*/
 dcb.fDtrControl = DTR_CONTROL_DISABLE; /* DTR flow control type - assert DTR*/
 dcb.fDsrSensitivity = FALSE;      /* DSR sensitivity*/
 dcb.fTXContinueOnXoff = FALSE/*TRUE*/;/* XOFF continues Tx - don't use XON/XOFF*/

 dcb.fOutX = FALSE;                /* XON/XOFF out flow control*/
```

```
 dcb.fInX = FALSE;                  /* XON/XOFF in flow control */
 dcb.fErrorChar = FALSE;            /* error replacement - off*/
 dcb.fNull = FALSE;                 /* null stripping - off*/
 dcb.fRtsControl = RTS_CONTROL_DISABLE; /*RTS_CONTROL_DISABLE;*/   /* RTS flow control */
 dcb.fAbortOnError = FALSE;         /* abort reads/writes on error*/
 /* dcb.fDummy2:17 is reserved  */
 dcb.wReserved = 0;                 /* not currently used*/
 dcb.XonLim = 0;                    /* transmit XON threshold*/
 dcb.XoffLim = 0;                   /* transmit XOFF threshold*/
 dcb.ByteSize = 8;                  /* number of bits/byte, 4-8*/
 dcb.Parity = NOPARITY;             /* 0-4=no,odd,even,mark,space*/
 dcb.StopBits = ONESTOPBIT;         /* 0,1,2 = 1, 1.5, 2*/
 dcb.XonChar = 0;                   /* Tx and Rx XON character*/
 dcb.XoffChar = 1;                  /* Tx and Rx XOFF character*/
 dcb.ErrorChar = 0;                 /* error replacement character*/
 dcb.EofChar = 0;                   /* end of input character*/
 dcb.EvtChar = 0;                   /* received event character*/
 /* don't use dcb.wReserved1 */



 portReady = SetCommState (handle, &dcb);

 if (portReady==1)
 {
  sprintf(traceString,"PLPTX: Port opened successfully");
  plptxTraceOutput(traceString);
 }

 SetupComm(handle, INPUT_BUFFER_LEN, OUTPUT_BUFFER_LEN);

 /* set port timeouts */

 timeoutsDefault.ReadIntervalTimeout         = MAXDWORD;   /* TO Do - change this.... */
 timeoutsDefault.ReadTotalTimeoutMultiplier  = 0;
 timeoutsDefault.ReadTotalTimeoutConstant    = 0;
 timeoutsDefault.WriteTotalTimeoutMultiplier = 0;
 timeoutsDefault.WriteTotalTimeoutConstant   = 0;

 return handle;
}
/*****************************************************************************************/
/*   plpbuCreateThreads
/*
/* Create the threads for transmitting, receiving */
/*
/*****************************************************************************************/
void plpbuCreateThreads (void)        /* TO DO - ought to make this function return TRUE if successful, FALSE if fail
as in hubu_fnc.c*/
{
 Int32    threadId;

 /* thread for transmit */


 if ( plpbuContext.txHandle == INVALID_HANDLE_VALUE )
 {
```



```
  /*Create the thread for transmitting*/
  plpbuContext.txHandle = CreateThread ( NULL, /*security attributes */
                                0,   /*Stack size */
                                transmitPacket, /* Tx Thread function*/
                                0,   /*Parameter */
                                0,   /*Create flags */
                                &threadId /*Thread identifier*/
                                );
 }

 if ( plpbuContext.rxHandle == INVALID_HANDLE_VALUE )
 {
  /*Create the thread for receiving*/
  plpbuContext.rxHandle = CreateThread ( NULL, /*security attributes */
                                0,   /*Stack size */
                                plpbuReceivePacket, /* Rx Thread function*/
                                0,   /*Parameter */
                                0,   /*Create flags */
                                &threadId /*Thread identifier*/);
 }


 if ( (plpbuContext.rxHandle == INVALID_HANDLE_VALUE) ||
        (plpbuContext.txHandle == INVALID_HANDLE_VALUE) )
 {

  if  (plpbuContext.txHandle == INVALID_HANDLE_VALUE)
  {
   DevFail ("PLPTX: tx thread creation failed");
  }
  else
  {
   DevFail ("PLPTX: rx thread creation failed");
  }
 }
#if defined (DEVELOPMENT_VERSION)
  if (plpbuContext.txHandle != 0)
  {
   sprintf(traceString,"PLPTX: tx thread created successfully");  /* thread appears not to be created if this isn't
present....*/
   plptxTraceOutput(traceString);
  }

  if (plpbuContext.rxHandle != 0)
  {
   sprintf(traceString,"PLPTX: rx thread created successfully");  /* thread appears not to be created if this isn't
present....*/
   plptxTraceOutput(traceString);
  }
#endif
}
/*****************************************************************************************
* plpbuLockQueues                                                *
*                                                                *
*  Locks transmission Queues                                     *
*                                                                *
*****************************************************************************************/
static void plpbuLockQueues(void)
```

```
{
 EnterCriticalSection (&plpbuContext.qLock);
}

/*****************************************************************************
* plpbuUnlockQueues                                                          *
*                                                                            *
*  Locks transmission Queues                                                 *
*                                                                            *
*****************************************************************************/
static void plpbuUnlockQueues(void)
{
 LeaveCriticalSection (&plpbuContext.qLock);
}

/*****************************************************************************
* plpbuSignalMoreTxData                                                      *
*                                                                            *
* signals that there is more data to send                                    *
*                                                                            *
*****************************************************************************/
static void plpbuSignalMoreTxData (void)
{
 SetEvent (plpbuContext.txSignal);
#if defined (DEBUG_PLPTX)
 sprintf(traceString,"PLPTX: signalled more data");
 plptxTraceOutput(traceString);
#endif
}
/*****************************************************************************
* plptxBusWriteData                                                          *
*                                                                            *
*  places pkt on transmit Queue                                              *
*                                                                            *
*****************************************************************************/
void plptxBusWriteData (SignalBuffer  *signalBuffer)
{
 DevAssert (*signalBuffer ->type = SIG_PLPTX_BUS_WRITE_DATA_REQ);

 plpbuPlaceTxDataOnQueue (signalBuffer);
 plpbuSignalMoreTxData () ;

#if defined (DEBUG_PLPTX)
 sprintf(traceString,"PLPTX: entered BusWriteData");
 plptxTraceOutput(traceString);
#endif

 KiDestroySignal (signalBuffer);
}

/*****************************************************************************
* transmitPacket                                                             *
*                                                                            *
*  Thread that continues to transmit as long as there's data to send         *
*                                                                            *
*****************************************************************************/
DWORD WINAPI  transmitPacket (LPVOID ptr)
{
```



```
 /* variables */
 Boolean       dataToSend;
 Boolean       result = TRUE;
 Int8*         txPtr;
 Int16         txLenToSend;
 SignalBuffer    signalBuffer = kiNullBuffer;

 PARAMETER_NOT_USED (ptr);

#if defined (DEDUG_PLPTX)
 sprintf(traceString,"PLPTX: entered transmit packet");
 plptxTraceOutput(traceString);


 if (result == FALSE)
 {
  sprintf(traceString,"PLPTX: transmit packet - result is FALSE");
  plptxTraceOutput(traceString);
 }
#endif

 while ( result== TRUE)
 {
#if defined (DEBUG_PLPTX)
  if (result ==TRUE)
  {
   sprintf(traceString,"PLPTX: transmit packet, while loop - result is TRUE");
   plptxTraceOutput(traceString);
  }

  sprintf(traceString,"PLPTX: in transmit packet while loop");
  plptxTraceOutput(traceString);
#endif

  WaitForSingleObject (plpbuContext.txSignal, INFINITE);  /* timeout is infinite */

  /* main thread has signalled that there is more data to tx
    loop reading from the queue til its empty or write fails */

  do
  {

   /* read data off the queue */

   if ( ( dataToSend = plpbuTakeTxDataFromQueue (&signalBuffer) ) == TRUE)
   {
#if defined (DEDUG_PLPTX)
    sprintf(traceString,"PLPTX: writing data to serial port");
    plptxTraceOutput(traceString);
#endif
    txPtr = (Int8 *) &signalBuffer.sig->plptxBusWriteDataReq.txData;
    txLenToSend = signalBuffer.sig->plptxBusWriteDataReq.txDataSize;

    WriterGeneric ( txPtr, txLenToSend);

    KiDestroySignal ( &signalBuffer);
   }
   else
```

```
    }
#if defined (DEBUG_PLPTX)
      sprintf(traceString,"PLPTX: transmit packet no data in Q  so didn't enter if loop");
      plptxTraceOutput(traceString);
#endif
    }
   } while ((dataToSend == TRUE) && (result==TRUE));
  }
 return 0;
}

/*****************************************************************************
 * Function: WriterGeneric
 * Parameters: lpBuf     Data to write
 *          dwToWrite  Number of bytes to write
 *
 * Description:
 * Write the number bytes to the COM port
 * ***************************************************************************/

void WriterGeneric(Int8 * lpBuf, DWORD dwToWrite)
{
 OVERLAPPED osWrite = {0};
 HANDLE hArray[1];
 DWORD dwWritten;
 DWORD dwRes;
 int bytesWritten = 0;
 int packetBytesWritten = 0;

 osWrite.hEvent = plpbuContext.wrEvent;
 hArray[0] = plpbuContext.wrEvent;

 /*
 * issue write   */

while (bytesWritten != dwToWrite)
{
 if ( !WriteFile(plpbuContext.ioHandle, (lpBuf + packetBytesWritten), dwToWrite, &dwWritten, &osWrite ))
  {

   if (GetLastError() == ERROR_IO_PENDING)
   {
    /*
     * write is delayed
    */
#if defined (DEBUG_PLPTX)
    sprintf(traceString,"write is delayed in writerGeneric");
    plptxTraceOutput(traceString);
#endif
    dwRes = WaitForSingleObject ( hArray[0], INFINITE);

    switch(dwRes)
    {
     /*
      * write event set
      */
     case WAIT_OBJECT_0:
      SetLastError(ERROR_SUCCESS);
```



```
      if (!GetOverlappedResult( plpbuContext.ioHandle, &osWrite, &dwWritten, FALSE))
      {
#if defined (DEBUG_PLPTX)
       sprintf(traceString,"WriterGeneric -GetOverlappedresult, dwWritten is: %d", dwWritten);
       plptxTraceOutput(traceString);
       sprintf(traceString,"WriterGeneric -GetOverlappedresult, dwToWrite is: %d", dwToWrite);
       plptxTraceOutput(traceString);
#endif
       if (GetLastError() == ERROR_OPERATION_ABORTED)
       {
        DevFail("Write aborted\r\n");
       }
       else
       {
        /* LPVOID lpMsgBuf;   */
#if defined (DEBUG_PLPTX)
        sprintf(traceString,"hello");
        plptxTraceOutput(traceString);
#endif
/*****************************************************************************************************
************************/
#if 0  /* displays windows error - also tends to crash the pc after displaying it - quite useful though!*/
                                        FormatMessage(
      FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
                                        NULL,
                                        GetLastError(),
                                        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default
language
                                        (LPTSTR) &lpMsgBuf,
                                                0,
                                        NULL
                                                );

                                                // Display the string.
                                                MessageBox( NULL, lpMsgBuf, "GetLastError",
MB_OK|MB_ICONINFORMATION|MB_SYSTEMMODAL                 );

                                                // Free the buffer.
                                                LocalFree( lpMsgBuf );
#endif
/*****************************************************************************************************
************************/
         DevFail("GetOverlappedResult(in Writer)");
        }
       }
       else
       {
        packetBytesWritten = packetBytesWritten + dwWritten;
        dwToWrite = dwToWrite - dwWritten;
        bytesWritten = 0;
       }

  /*    if (dwWritten != dwToWrite)
       {
        DevFail ("Error writing data to port (overlapped)");
        break;
       }            */
```

```
#if defined (DEBUG_PLPTX)
        sprintf(traceString,"WriterGeneric - dwWritten is: %d", dwWritten);
        plptxTraceOutput(traceString);
        sprintf(traceString,"WriterGeneric - dwToWrite is: %d", dwToWrite);
        plptxTraceOutput(traceString);
#endif
        break;

      case WAIT_TIMEOUT:

      case WAIT_FAILED:

      default:
       DevFail ("WaitForMultipleObjects (WriterGeneric)");
       break;
    }
   }
   else
   {
    /*
     * writefile failed, but it isn't delayed
     */
    DevFail ("WriteFile (in Writer)");
   }
  }

  else
  {
   /*
    * writefile returned immediately
    */
   sprintf(traceString,"writefile returned immediately in writerGeneric");
   plptxTraceOutput(traceString);

   packetBytesWritten = packetBytesWritten + dwWritten;
   dwToWrite = dwToWrite - dwWritten;
   bytesWritten = 0;

   if (dwWritten != dwToWrite)
   {
    DevFail ("Write timed out. (immediate)\r\n");
   }
  }
}
 ResetEvent (osWrite.hEvent);
 ResetEvent (plpbuContext.txSignal);

 return;
}


/*****************************************************************************
 * Function: plpbuProcessRxData
 * Parameter:
 *         rxLen  Number of bytes received.
 * Description:
 * Called from the interrupt routine, indicates the number of bytes received.
 * If we have received all we are waiting for process the state otherwise just
```



```
 * wait until the requisite amount of data has been received.
 *****************************************************************************/
static void plpbuProcessRxData ( Int16 rxLen)
{
 Int8 packetType;
 Int8 signalName;
 Int8 signalAcked;

 SignalBuffer signalToSend = kiNullBuffer;
 int nameCount;
 int keyCount;
 Int8 rxByte; /* received byte */

 plpbuContext.rxLenReceived += rxLen;

 if ( plpbuContext.rxLenReceived == plpbuContext.rxLenWaiting)
 {
  switch (plpbuContext.rxState)
  {
   case PLPTX_BUS_START:
    rxByte = GET_INT8(0, plpbuContext.rxPtr);
    if (rxByte == PRE_AMBLE_BYTE)
    {
     plpbuContext.startByteCounter++;
    }

    /* if rxByte != PRE_AMBLE_BYTE ignore that bit and go to check the next one */

    if (plpbuContext.startByteCounter == 3) /* there should be 3 pre amble bytes.... */
    {
     plpbuContext.rxLenWaiting  = BDADDR_SIZE;
     plpbuContext.rxPtr         = &plpbuContext.rxBuffer[0];
     plpbuContext.rxLenReceived = 0;
     plpbuContext.rxState       = PLPTX_BUS_RX_REMOTE_DEVICE_ID;      /* having received all of the pre-
amble, change to the next state */
     plpbuContext.startByteCounter =0;
    }
    else
    {
     plpbuContext.rxLenWaiting  = 1;   /* ready to receive pre-amble byte */
     plpbuContext.rxPtr         = &plpbuContext.rxBuffer[0];
     plpbuContext.rxLenReceived = 0;
     plpbuContext.rxState       = PLPTX_BUS_START;
     plptxContext.busConnected = FALSE;
    }

    break;

   case PLPTX_BUS_RX_REMOTE_DEVICE_ID:
    GET_BDADDR(0, plpbuContext.rxPtr, plptxContext.rxDeviceId);

    if( memcmp(&plptxContext.rxDeviceId, &plptxContext.localDeviceId, BDADDR_SIZE) == 0)   /* signal
received was sent by same device, i.e.invalid */
    {
     /* want to throw away the received data */
     plpbuContext.rxLenWaiting  = 1;   /* ready to receive pre-amble byte */
     plpbuContext.rxPtr         = &plpbuContext.rxBuffer[0];
     plpbuContext.rxLenReceived = 0;
```

```
        plpbuContext.rxState      = PLPTX_BUS_START;
        plptxContext.busConnected = FALSE;
#if defined (DEVELOPMENT_VERSION)
        sprintf(traceString,"Received signal with my own ID - ignoring it");
        plptxTraceOutput(traceString);
#endif
       }
       else
       {
        plpbuContext.rxLenWaiting  = 1;   /* ready to receive type byte */
        plpbuContext.rxPtr         = &plpbuContext.rxBuffer[0];
        plpbuContext.rxLenReceived = 0;
        plpbuContext.rxState       = PLPTX_BUS_RX_PACKET_TYPE;

       }
      break;


     case PLPTX_BUS_RX_PACKET_TYPE:
      packetType = GET_INT8(0,plpbuContext.rxPtr);
#if defined (DEBUG_PLPTX)
      sprintf(traceString,"packetType is : %c", packetType);
      plptxTraceOutput(traceString);
#endif
      switch(packetType)
      {
       case INFO_TYPE:
#if defined (DEVELOPMENT_VERSION)
         sprintf(traceString,"Info type received");
         plptxTraceOutput(traceString);
#endif

         plpbuContext.rxLenWaiting = (PLPTX_BUS_OUT_INFO_DATA_SIZE - 1);
         plpbuContext.rxPtr = &plpbuContext.rxBuffer [0];
         plpbuContext.rxLenReceived = 0;
         plpbuContext.rxState = PLPTX_BUS_RX_SIGNAL;
        break;

       case ACK_TYPE:   /* NB signal received could be an ACK or a NACK */
#if defined (DEVELOPMENT_VERSION)
         sprintf(traceString,"Ack/nack type received");
         plptxTraceOutput(traceString);
#endif

         plpbuContext.rxLenWaiting = (PLPTX_BUS_ACK_TYPE_SIZE - 1);
         plpbuContext.rxPtr = &plpbuContext.rxBuffer [0];
         plpbuContext.rxLenReceived = 0;
         plpbuContext.rxState = PLPTX_BUS_RX_SIGNAL;
        break;

       case START_TYPE:
#if defined (DEVELOPMENT_VERSION)
         sprintf(traceString,"start type received");
         plptxTraceOutput(traceString);
#endif

         plpbuContext.rxLenWaiting = (PLPTX_BUS_START_SEQUENCE_SIZE - 1);
         plpbuContext.rxPtr = &plpbuContext.rxBuffer [0];
```



```
         plpbuContext.rxLenReceived = 0;
         plpbuContext.rxState = PLPTX_BUS_RX_SIGNAL;
        break;

       default:
         /* invalid signal, so ignore it and return to the START state */
         plpbuContext.rxState = PLPTX_BUS_START;
        break;
       }
      break;

     case PLPTX_BUS_RX_SIGNAL:
      signalName = GET_INT8 (0, plpbuContext.rxPtr);

#if defined (DEBUG_PLPTX)
      sprintf(traceString,"signalName is : %c", signalName);
      plptxTraceOutput(traceString);
#endif

      switch (signalName)
      {
       case START_SIGNAL:
#if defined (DEVELOPMENT_VERSION)
         sprintf(traceString,"start signal received");
         plptxTraceOutput(traceString);
#endif
         if (plptxContext.start1Sent = TRUE)
         {
         /* activate start sequence (part 2 of 2)*/
         KiCreateSignal  (SIG_PLPTX_START_SEQUENCE2_REQ,
                  sizeof (PlptxStartSequence2Req),
                  &signalToSend);

         signalToSend.sig->plptxStartSequence2Req.myTaskId = PLPTX_TASK_ID;

         KiSendSignal  (PLPTX_TASK_ID, &signalToSend);
         }
         else
         {
#if defined (DEVELOPMENT_VERSION)
         sprintf(traceString,"no start2req sent as haven't sent start1 yet!");
         plptxTraceOutput(traceString);
#endif
         }
         plptxContext.start1Received = TRUE;
         plpbuContext.rxState = PLPTX_BUS_START;
        break;

       case START2_SIGNAL:
#if defined (DEVELOPMENT_VERSION)
         sprintf(traceString,"start2 signal received");
         plptxTraceOutput(traceString);
#endif
         if (plptxContext.start1Received == TRUE)
         {
         plptxStopStartSequenceTimer ();

         plptxContext.plptxState = CONNECTED;
```

```
        /* send plpStartSequenceCnf internally to PLP task*/
        KiCreateSignal  (SIG_PLPTX_START_SEQUENCE_CNF,
                sizeof (PlptxStartSequenceCnf),
                &signalToSend);

        signalToSend.sig->plptxStartSequenceCnf.myTaskId = PLPTX_TASK_ID;

        KiSendSignal  (PLP_TASK_ID, &signalToSend);
        }
        else
        {
#if defined (DEVELOPMENT_VERSION)
        sprintf(traceString,"startSequenceCnf not sent as start1 not received, only start2");
        plptxTraceOutput(traceString);
#endif
        }
        plpbuContext.rxState = PLPTX_BUS_START;

       break;

      case PLPTX_BUS_OUT_INFO:
#if defined (DEVELOPMENT_VERSION)
        sprintf(traceString,"PLPTX_BUS_OUT_INFO signalName received");
        plptxTraceOutput(traceString);
#endif

        KiCreateSignal (SIG_PLPTX_IN_INFO_IND,
                sizeof (PlptxInInfoInd),
                &signalToSend);

        signalToSend.sig->plptxInInfoInd.myTaskId = PLPTX_TASK_ID;
        signalToSend.sig->plptxInInfoInd.plpStatus = PLP_COMMAND_OK;

        GET_BDADDR ((SIZE_TO_BDADDR - PLPTX_BUS_HEADER_SIZE - SIGNAL_TYPE_SIZE),
plpbuContext.rxPtr, signalToSend.sig->plptxInInfoInd.plpBtBdAddr);

        signalToSend.sig->plptxInInfoInd.plpFriendlyName.nameLen = GET_INT8 ((SIZE_TO_NAME_LEN -
PLPTX_BUS_HEADER_SIZE - SIGNAL_TYPE_SIZE), plpbuContext.rxPtr);

        for (nameCount = 0; nameCount<248; nameCount++)
        {
         signalToSend.sig->plptxInInfoInd.plpFriendlyName.name[nameCount] = GET_INT8 ((SIZE_TO_NAME -
PLPTX_BUS_HEADER_SIZE - SIGNAL_TYPE_SIZE + nameCount), plpbuContext.rxPtr);
        }

        for (keyCount = 0; keyCount<16; keyCount++)
        {
         signalToSend.sig->plptxInInfoInd.plpLinkKey[keyCount] = GET_INT8 ((SIZE_TO_LINK_KEY -
PLPTX_BUS_HEADER_SIZE - SIGNAL_TYPE_SIZE+ keyCount), plpbuContext.rxPtr);
        }

        KiSendSignal (PLP_TASK_ID,&signalToSend);
        plpbuContext.rxState = PLPTX_BUS_START;

        break;

       case PLPTX_BUS_ACK:
```



```
#if defined (DEVELOPMENT_VERSION)
        sprintf(traceString,"PLPTX_BUS_ACK signalName received");
        plptxTraceOutput(traceString);
#endif
        signalAcked = GET_INT8 (1, plpbuContext.rxPtr);

        switch (signalAcked)
        {
         case PLPTX_BUS_OUT_INFO:
#if defined (DEVELOPMENT_VERSION)
          sprintf(traceString,"PLPTX_BUS_OUT_INFO Ack received");
          plptxTraceOutput(traceString);
#endif
          KiCreateSignal (SIG_PLPTX_OUT_INFO_CNF,
                sizeof (PlptxOutInfoCnf),
                &signalToSend);

          signalToSend.sig -> plptxOutInfoCnf.myTaskId = PLPTX_TASK_ID;
          signalToSend.sig -> plptxOutInfoCnf.plpStatus = PLP_COMMAND_OK;

          KiSendSignal (PLP_TASK_ID, &signalToSend);
          plpbuContext.rxState = PLPTX_BUS_START;
          break;

         case PLPTX_FINISH_REQ_ACK:
#if defined (DEVELOPMENT_VERSION)
          sprintf(traceString,"PLPTX_FINISH_REQ_ACK signalName received");
          plptxTraceOutput(traceString);
#endif
          KiCreateSignal (SIG_PLPTX_IN_FINISH_IND,
                sizeof (PlptxInFinishInd),
                &signalToSend);

          signalToSend.sig -> plptxInFinishInd.myTaskId = PLPTX_TASK_ID;

          KiSendSignal (PLP_TASK_ID, &signalToSend);
          plpbuContext.rxState = PLPTX_BUS_START;
          break;

         default:
          /* invalid signal, so ignore it and return to the START state */
          plpbuContext.rxState = PLPTX_BUS_START;
          break;
        }
        break;

       default:
        /* invalid signal, so ignore it and return to the START state */
        plpbuContext.rxState = PLPTX_BUS_START;
        break;
      }
      plpbuContext.rxLenWaiting = 1;  /* size of type Byte */
      plpbuContext.rxPtr = &plpbuContext.rxBuffer [0];
      plpbuContext.rxLenReceived = 0;
      break;

     default:
      DevFail ("unrecognised state");
```

```
      break;
    }
  }
  else
  {
    /*Not read all we need yet, advance pointer to get the next bit*/
    plpbuContext.rxPtr += rxLen;

  }

}

/*****************************************************************************
 * Function: plpbuReceivePacket
 *
 * Description:
 * Thread for receiving data will continue to process.
 * ***************************************************************************/
DWORD WINAPI plpbuReceivePacket ( LPVOID ptr )
{

 Int32 receivedLen = 0;
 Boolean result    = 0;

 PARAMETER_NOT_USED (ptr);

if (plpbuContext.rxActive = TRUE)
{
 /*Read forever until a read error occurs*/
 while ( result == 0 )
 {
  result = ReadGeneric ( plpbuContext.rxPtr,   /*Buffer for data */
                (plpbuContext.rxLenWaiting - plpbuContext.rxLenReceived), /*Number of bytes to read*/
                &receivedLen        /*Number of bytes read*/ );

#if defined (DEBUG_PLPTX)
   sprintf(traceString,"receivedLen is : %d", receivedLen);
   plptxTraceOutput(traceString);
#endif

   DevAssert ( result == 0 );


   if (result == 0)
   {
    plpbuProcessRxData((Int16) receivedLen);
   }

 }
}
else
{
#if defined (DEVELOPMENT_VERSION)
 sprintf(traceString,"Read deactivated");
 plptxTraceOutput(traceString);
#endif
 }
 return 1;
```



```
}
/*****************************************************************************
 * Function: ReadGeneric
 * Parameters: lpBuf    Buffer to read data into
 *          dwToRead  Number of bytes to read
 *          readBytes Number of bytes actually read
 *
 * Description:
 * Reads from COMM port, if read does not complete immediately it will
 * wait for completion and then return.
 * ***************************************************************************/

            /* buffer for data*/                /*#bytes read*/
           /* plpbuContext.rxPtr*/ /* # bytes to read *//* &receivedLen*/
WtErr WINAPI ReadGeneric( Int8* lpBuf,        Int16 dwToRead, Int32 *readBytes )
{
  OVERLAPPED osReader = {0};  /* overlapped structure for read operations*/
  HANDLE    hArray[1];

  DWORD     dwRead;        /* bytes actually read */
  DWORD     dwRes;         /* result from WaitForSingleObject */
  BOOL      fWaitingOnRead = FALSE; /* just added ...setting this to FALSE */
  WtErr     retVal;
  /*
   * create  overlapped structure for read events
   */
  osReader.hEvent = plpbuContext.rdEvent;
  hArray[0] = osReader.hEvent;


  /*
   * Read from the COM port
   */

 /* if (!ReadFile (plpbuContext.ioHandle,lpBuf, 3, &dwRead, &osReader))          */
 if ( !ReadFile(plpbuContext.ioHandle, lpBuf, dwToRead, &dwRead, &osReader) )
  {

   if (GetLastError() != ERROR_IO_PENDING)   /* read not delayed? */
   {
     LPVOID lpMsgBuf;
    /* displays windows error */
                                        FormatMessage(
      FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
                                        NULL,
                                        GetLastError(),
                                        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default
language
                                        (LPTSTR) &lpMsgBuf,
                                                0,
                                        NULL
                                                );

                                                // Display the string.
                                                MessageBox( NULL, lpMsgBuf, "GetLastError",
MB_OK|MB_ICONINFORMATION|MB_SYSTEMMODAL                 );

                                                // Free the buffer.
```

```
                    LocalFree( lpMsgBuf );

   DevFail ("ReadFile in ReaderAndStatusProc");
   fWaitingOnRead = FALSE;
   /*retVal = WT_RS232_ERROR;   */
  }
  else
  {
   fWaitingOnRead = TRUE;
  }
 }
 else
 {
  /*Read returned immediately with some data */
  *readBytes = dwRead;
  fWaitingOnRead = FALSE;
  retVal = WT_OK;
 }

 /*
  * wait for pending operations to complete
  */
 if ( fWaitingOnRead )
 {
  dwRes = WaitForSingleObject( hArray [0],  INFINITE);
  switch(dwRes)
  {
   /*
    * read completed
    */
   case WAIT_OBJECT_0:
   if (!GetOverlappedResult(plpbuContext.ioHandle, &osReader, &dwRead, FALSE))
   {
    if (GetLastError() == ERROR_OPERATION_ABORTED)
    {
     DevFail("Read aborted\r\n");
    }
    else
    {
     DevFail("GetOverlappedResult (in Reader)");
    }

    /*Wait for event has failed*/
    retVal = WT_RS232_ERROR;
   }
   else
   {
     if ( dwRead == 0 )
     {
      /* Timed out before reading any data */
      retVal = WT_RECEIVE_TIMEOUT;
     }
     else
     {
      /*Read has completed, return numberofbytes read*/
      *readBytes = dwRead;
      retVal = WT_OK;
     }
```



```
     }

     break;


     case WAIT_TIMEOUT:
     default:
      DevFail("WaitForMultipleObjects(Reader & Status handles)");
      retVal = WT_RS232_ERROR;
      break;
    }
   }

   ResetEvent (osReader.hEvent);

   return retVal;
}

/*****************************************************************************
* plpbuTakeTxDataFromQueue
*
*  Removes packet from the Queue to send.
*
*****************************************************************************/
static Boolean plpbuTakeTxDataFromQueue (SignalBuffer  *signalBuffer)
{
 Boolean retVal = FALSE;

 /* lock the queues */
 plpbuLockQueues ();

 /* Get Data from the queues */
 if ( KiOnQueue (&plpbuContext.cmdQueue) == TRUE)
 {
  KiDequeue(&plpbuContext.cmdQueue, signalBuffer);
  retVal = TRUE;
 }

 /* finished with the Queues */
 plpbuUnlockQueues ();

 return retVal;

}


/*****************************************************************************
* plpbuPlacetxDataOnQueue
*
*  places data to be sent on the queue
*
*****************************************************************************/
static void plpbuPlaceTxDataOnQueue (SignalBuffer *signalBuffer)
{
 /* lock queues */
 plpbuLockQueues ();

 /* put data on queue */
```

```
 KiEnqueue (&plpbuContext.cmdQueue, signalBuffer);

 /* finished with queues - unlock */
 plpbuUnlockQueues ();
}
/*****************************************************************************/
```



```
/*****************************************************************************
 * * $Workfile:   plptxbu_fnc.h  $
 * $Revision:
 *     $Date:
*****************************************************************************
 *
 *  Designed by    : PKR
 *  Coded by       :
 *  Tested by      : PKR
 *
*****************************************************************************
 *
 * File Description
 * ----------------
 *  Pairing Link Protocol - Main function prototypes
 *
 *****************************************************************************/

#if !defined (PLPTXBU_FNC_H)
# define PLPTXBU_FNC_H


  #if defined (PLPTX_TRACE_OUTPUT)
    void  plptxTraceOutput          (char*);
  #endif

#endif
```

```
/*****************************************************************************
 *
 * $Workfile: plptx_typ.h
 * $Revision:
 *    $Date:
 *
*****************************************************************************
 *
 * File Description
 * ----------------
 *
 *****************************************************************************/

#if !defined (PLPTX_TYP_H)
#define     PLPTX_TYP_H

/*****************************************************************************
 * Nested Include Files
 *****************************************************************************/
#if !defined (KERNEL_H)
#  include "kernel.h"
#endif

#if !defined (BT_TYP_H)
#  include "bt_typ.h"
#endif

#if !defined (TM_TYP_H)
#  include "tm_typ.h"
#endif

/*****************************************************************************
 * Manifest Constants
 ********************/
#define PLPTX_START_SEQUENCE_TIMER_VALUE 500 /* was 500 */

/*****************************************************************************
 * Types used in Prototypes and Globals
 ********************/

typedef struct PlptxBufferTag
{
 Int8   txData[512];
} PlptxBuffer;

#endif /* of !defined (HU_TYP_H) */

/* END OF FILE */
```



```
/*****************************************************************************
 *
 * $Workfile:  plptxbu_typ.h $
 * $Revision:
 *    $Date:
 *
*****************************************************************************
 *
 * File Description
 * ----------------
 * Globaly useful bus task functions/variables.
 *
 *****************************************************************************/

#if !defined (PLPTXBU_TYP_H)
#define     PLPTXBU_TYP_H

/*****************************************************************************
 * Types used in Prototypes and Globals
 *****************************************************************************/
#define PC_COM_PORT  "COM2"
#define INPUT_BUFFER_LEN  1024
#define OUTPUT_BUFFER_LEN 1024
#define PC_BUS_BAUD_RATE 9600 /*was 9600 */

#define SIGNAL_TYPE_SIZE 1
#define SIGNAL_NAME_SIZE 1
#define BDADDR_SIZE 6

#define NAME_LEN_SIZE 1
#define NAME_SIZE 248
#define PLPTX_BUS_HEADER_SIZE (3 + BDADDR_SIZE)

#define SIZE_TO_SIGNAL_NAME  (PLPTX_BUS_HEADER_SIZE + SIGNAL_TYPE_SIZE)     /* 1 */
#define SIZE_TO_BDADDR      (PLPTX_BUS_HEADER_SIZE + SIGNAL_TYPE_SIZE +
SIGNAL_NAME_SIZE)  /* 2 */
#define SIZE_TO_NAME_LEN    (PLPTX_BUS_HEADER_SIZE + SIGNAL_TYPE_SIZE +
SIGNAL_NAME_SIZE + BDADDR_SIZE)    /* 8 */
#define SIZE_TO_NAME        (PLPTX_BUS_HEADER_SIZE + SIGNAL_TYPE_SIZE + SIGNAL_NAME_SIZE +
BDADDR_SIZE + NAME_LEN_SIZE)  /*  9  */
#define SIZE_TO_LINK_KEY    (PLPTX_BUS_HEADER_SIZE + SIGNAL_TYPE_SIZE +
SIGNAL_NAME_SIZE + BDADDR_SIZE + NAME_LEN_SIZE + NAME_SIZE ) /*257*/

#define SIZE_TO_TYPE PLPTX_BUS_HEADER_SIZE
#define SIZE_TO_SIG_NAME (1 + PLPTX_BUS_HEADER_SIZE)
#define SIZE_TO_SIGNAL_BEING_ACKED (2 + PLPTX_BUS_HEADER_SIZE)


#define SIZE_TO_SIGNAL (2 + PLPTX_BUS_HEADER_SIZE)

#define INFO_TYPE  90
#define ACK_TYPE   100
#define START_TYPE 65 /* "A" */
#define PRE_AMBLE_BYTE 37    /* "%" */

#define PLPTX_BUS_OUT_INFO 200
#define PLPTX_BUS_ACK   210
#define PLPTX_FINISH_REQ_ACK 220
```

```
#define START_SIGNAL_NAME 65 /* "A" */
#define START2_SIGNAL_NAME 66 /* "B" */

#define START_SIGNAL 65 /* "A" */
#define START2_SIGNAL  66 /* "B" */

#define PLPTX_BUS_ACK_TYPE_SIZE 3

#define PLPTX_BUS_OUT_INFO_DATA_SIZE 273
#define PLPTX_BUS_START_SEQUENCE_SIZE 3

#endif

/* END OF FILE */
```



```
/*****************************************************************************
 *
 * $Workfile:  plptx_sig.h $
 * $Revision:
 *    $Date:
 *
 *****************************************************************************
 *
 * Designed by   : PKR
 * Detailed Design:
 * Coded by      : PKR
 * Tested by     :
 *
 *****************************************************************************
 *
 * File Description
 * ----------------
 *  PLPTX signal definitions
 *
*****************************************************************************/

#if !defined (PLPTX_SIG_H)
#define PLPTX_SIG_H
#endif

#if !defined (SYSTEM_H)
# include "system.h"
#endif

#if !defined (HCI_TYP_H)
#include "hci_typ.h"
#endif

#if !defined (PLP_TYP_H)
#include "plp_typ.h"
#endif

#if !defined (PLPTX_TYP_H)
#include "plptx_typ.h"
#endif

/*****************************************************************************
 * Type Definitions
 *****************************************************************************/
typedef struct  PlptxTestTag {
 Int8           timeout;
}PlptxTest;

typedef struct  PlptxInInfoIndTag {
 TaskId         myTaskId;
 PlpStatus      plpStatus;
 BtBdAddr       plpBtBdAddr;
 PlpFriendlyName  plpFriendlyName;
 Int8           plpLinkKey [BT_ENCRYPTION_KEY_SIZE];
}PlptxInInfoInd;

typedef struct  PlptxInInfoRspTag {
 TaskId         myTaskId;
```

```
 PlpStatus        plpStatus;
}PlptxInInfoRsp;

typedef struct   PlptxOutInfoReqTag {
 TaskId           myTaskId;
 PlpStatus        plpStatus;
 BtBdAddr         plpBtBdAddr;
 PlpFriendlyName  plpFriendlyName;
 Int8             plpLinkKey [BT_ENCRYPTION_KEY_SIZE];
}PlptxOutInfoReq;

typedef struct   PlptxOutInfoCnfTag {
 TaskId           myTaskId;
 PlpStatus        plpStatus;
}PlptxOutInfoCnf;

typedef struct PlptxBusWriteDataReqTag
{
 Int16            txDataSize;
 PlptxBuffer      txData;
}PlptxBusWriteDataReq;

typedef struct PlptxBusAckReqTag
{
 TaskId           myTaskId;
 Int8             type;
 Int8             signalName;
 Int8             signalBeingAcked;
}PlptxBusAckReq;

typedef struct PlptxOutFinishReqTag
{
 TaskId           myTaskId;
}PlptxOutFinishReq;

typedef struct PlptxInFinishIndTag
{
 TaskId           myTaskId;
}PlptxInFinishInd;

typedef struct PlptxStartSequenceReq
{
 TaskId           myTaskId;
 BtBdAddr         localBtBdAddr;
}PlptxStartSequenceReq;

typedef struct PlptxStartSequence2Req
{
 TaskId           myTaskId;
 BtBdAddr         localBtBdAddr;
}PlptxStartSequence2Req;

typedef struct PlptxStartSequenceCnf
{
 TaskId           myTaskId;
} PlptxStartSequenceCnf;

/* END OF FILE */
```



```
/*****************************************************************************
*
* $Workfile:   plptxsigbas.h $
* $Revision:
*    $Date:
*
*****************************************************************************
*
* File Description
* ----------------
*  Signal bases used by PAIRING LINK PROTOCOL Transport Task
*
*****************************************************************************
*
* Revision Details
* ----------------
*
* $Log:
*
*****************************************************************************/
 PLPTX_SIGNAL_BASE = LAST_PLP_SIGBASE + 0x0100,
 LAST_PLPTX_SIGBASE  = PLPTX_SIGNAL_BASE,
```

```
/*****************************************************************************
 *
 * $Workfile:  plptxsig.h $
 * $Revision:
 *    $Date:
 *
 *****************************************************************************
 *
 * File Description
 * ----------------
 *  PAIRING LINK PROTOCOL Signals used in Genie
 *
*****************************************************************************/

  SIG_DEF( SIG_PLPTX_DUMMY = PLPTX_SIGNAL_BASE,       EmptySignal                plptxDummy
)
  SIG_DEF( SIG_PLPTX_TEST,                        PlptxTest                plptxTest                )
  SIG_DEF( SIG_PLPTX_IN_INFO_IND,                 PlptxInInfoInd           plptxInInfoInd           )
  SIG_DEF( SIG_PLPTX_IN_INFO_RSP,                 PlptxInInfoRsp           plptxInInfoRsp           )
  SIG_DEF( SIG_PLPTX_OUT_INFO_REQ,                PlptxOutInfoReq          plptxOutInfoReq
)
  SIG_DEF( SIG_PLPTX_OUT_INFO_CNF,                PlptxOutInfoCnf          plptxOutInfoCnf
)
  SIG_DEF( SIG_PLPTX_BUS_WRITE_DATA_REQ,          PlptxBusWriteDataReq
plptxBusWriteDataReq          )
  SIG_DEF( SIG_PLPTX_BUS_ACK_REQ,                 PlptxBusAckReq           plptxBusAckReq
)
  SIG_DEF( SIG_PLPTX_OUT_FINISH_REQ,              PlptxOutFinishReq        plptxOutFinishReq
)
  SIG_DEF( SIG_PLPTX_IN_FINISH_IND,               PlptxInFinishInd         plptxInFinishInd         )
  SIG_DEF( SIG_PLPTX_START_SEQUENCE_REQ,          PlptxStartSequenceReq
plptxStartSequenceReq         )
  SIG_DEF( SIG_PLPTX_START_SEQUENCE2_REQ,         PlptxStartSequence2Req
plptxStartSequence2Req        )
  SIG_DEF( SIG_PLPTX_START_SEQUENCE_CNF,          PlptxStartSequenceCnf
plptxStartSequenceCnf         )
```



```
/*****************************************************************************
 *
 * $Workfile:  plptxsigun.h $
 * $Revision:
 *    $Date:
 *
 *****************************************************************************
 *
 * File Description
 * ----------------
 *  Signal bases used by PAIRING LINK PROTOCOL Transport Task
 *
 *****************************************************************************
#if !defined (DMSH_SIG_H)
#include "dmsh_sig.h"
#endif

#if !defined (DMIQ_SIG_H)
#include "dmiq_sig.h"
#endif

#if !defined (DMSC_SIG_H)
#include "dmsc_sig.h"
#endif

#if !defined (DMCN_SIG_H)
#include "dmcn_sig.h"
#endif

#if !defined (DMSO_SIG_H)
#include "dmso_sig.h"
#endif

#if !defined (DML2_SIG_H)
#include "dml2_sig.h"
#endif

#if !defined (DMSP_SIG_H)
#include "dmsp_sig.h"
#endif

#if !defined (DMIN_SIG_H)
#include "dmin_sig.h"
#endif

#if !defined (DMSD_SIG_H)
#include "dmsd_sig.h"
#endif

#if !defined (L2IF_SIG_H)
#include "l2if_sig.h"
#endif

#if defined (DM_TRACE_OUTPUT)
#include "emmi_sig.h"
#endif

#define SIGNAL TVSIGNAL
```

```
#if !defined (SIG_DEF_H)
#include "sig_def.h"
#endif

#if !defined (KERNEL_H)
#include "kernel.h"
#endif

#if !defined (PLPTX_SIG_H)
#include "plptx_sig.h"
#endif

union Signal
{

#if defined (PLPTX_SIGNALS)
  #include "plptxsig.h"
#endif

#if defined (DM_SIGNALS)
  #include "dmsig.h"
#endif

#if defined (PLPTX_TRACE_OUTPUT)
    TestFileOut     testFileOut;
#endif

  KiInitialiseTask  initialise;
  KiInitialiseTask  kiInitialiseTask;
  KiTimerExpiry     kiTimerExpiry ;

};
```